MASTER EN SCIENCES INFORMATIQUES

Specifying and verifying real-time systems with TRIO-PVS

Kohl, Denise

Award date:
1997

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

Specifying and Verifying Real-Time Systems with TRIO/PVS

Denise Kohl

Promoter: Prof. E. Dubois

Thesis submitted in conformity with the requirements for the degree of 'Licencié et Maître en Informatique'
1996-1997

RUE GRANDGAGNAGE, 21 • B - 5000 NAMUR (BELGIUM)

Acknowledgments

To all those who have contributed to the realization of my thesis. Specially, to my parents, who have given me the financial possibility to do my studies.

To Prof. Eric Dubois, my promoter, for his support and advice.

To Philippe Du Bois, François Chabot and the entire team for their advice and for reading draft parts of this work.

To Prof. Dino Mandrioli and Prof. Angelo Morzenti for having welcomed me in Milano and for the opportunity to prepare this work at the Politecnico.

To Angelo Garganti for his advice during my stay in Milano and for having helped me in improving my Italian.

To Pascal Bauler for reading a draft version of this work.

To Constant Backes, my future husband, for his moral support.

Abstract

Today it is widely accepted, that it is necessary to make a formal specification of the program first, before starting to implement it in a programming language.

In this thesis, we introduce the specification language TRIO+ and we propose a specification of a case study, i.e. an Energy_Meter. After a short introduction to the PVS theorem prover, we show how the encoding of TRIO into PVS can be useful for verifying real-time systems. We illustrate the use of this encoding through the falsification of the first part of the case study and the verification of the second part of the case study.

In this paper, we also give a specification of the case study in ALBERT and further, we compare the two specification languages, TRIO and ALBERT.

Résumé

De nos jours, il est largement admis qu'il est nécessaire de spécifier formellement un programme avant de l'implémenter dans un language de programmation.

Dans de document, nous introduisons le language de spécification TRIO+ et nous proposons une spécification d'une étude de cas, un Compteur d'Energie en TRIO+. Après une briève introduction au système de preuve PVS, nous montrons que l'encodage de TRIO en PVS peut être utile pour la vérification de système en temps réel. Nous illustrons l'emploi de cet encodage par la falsification de la première partie de l'étude de cas et la vérification de la deuxième partie de l'étude de cas.

Dans de papier, nous donnons aussi une spécification en ALBERT de l'étude de cas et, enfin nous comparons les deux langages de spécification, TRIO et ALBERT.

Contents

List of Figures

Introduction

Today it is widely accepted, that it is necessary to make a formal specification of the program first, before starting to implement it in a programming language. "Program specifications present WHAT the desired program is supposed to do, rather than HOW it is going to accomplish it" [2].

The validation of specifications are often done by human review and inspection, but specially for formal specifications it is possible to do much more, e.g. formal analysis, which can be mechanically checked [4].

In this work, we focus on the theorem prover [1] and proof checker [2] PVS and more specially on one of the encodings: TRIO in PVS.

PVS is a prototype for a system for formal specification and validation based on higher-order logic and TRIO is a first-order language to which temporal operators are added. The mechanical verification is "the process of performing logic/symbolic reasoning with the help of a computer program [16]. Two different approaches are used for this process:

- the finite-states methods
- the theorem prover.

Two of the different finite-state methods, the satisfiability checking and the model checking, are explained in [16].

The encoding of TRIO in PVS offers the opportunity, that a real-time specification can be verified by using the PVS proof checker. In Milan we had the interesting opportunity to use the encoding of TRIO+ in PVS to do the validation of the case study.

[1] A theorem prover often is a program, which employs a number of heuristics for verification and which is guided by the user [16].

[2] A proof checker requires that the user provides the ingenuity for carrying out the proof [16].

In the second chapter, we try to show how PVS can be useful in the validation of a specification. Further in this chapter, we give a PVS specification of the Airline Reservation System and we prove a number of properties of this system.

In the first chapter of this work, we give an introduction to TRIO, a specification language which was developed at the Politecnico di Milano.

In the third chapter, we explain briefly how TRIO+ is encoded in PVS and how a TRIO+ specification has to be translated to the encoding of TRIO+ in PVS before the specification can be validated by using the PVS proof checker.

The TRIO+ specification of the Energy_Meter case study is given in chapter four. In the last chapter, we show the validation of this case study by using the encoding of TRIO in PVS. First, we give a falsification of the first part of the case study, an Energy_Meter with one photocell, and a verification of the second part, an Energy_Meter with two photocells. Furthermore, we prove a property, particular situation the Energy_Meter can be faced to.

In chapter five, we propose an ALBERT specification of the same case study. ALBERT is a specification language developed at the University of Namur. In the last section of this chapter, we compare the two specification languages.

In the conclusion, we show, that as a further work, it is interesting to develop an encoding of ALBERT-CORE in PVS, to give the ALBERT language a tool for the validation of an ALBERT specification, which is partially mechanically.

Chapter 1

Introduction to TRIO

1.1 The TRIO Language

The TRIO Language is a logical language for the formal specification of real-time systems based on first-order logic [1], to which temporal operators are added that provide a metric on time, making it suitable for describing real-time properties [10]. So a formula can be true or false depending on the current evaluation of a given time t, which is left implicit.

Unlike most first-order languages, TRIO is typed. In a typed language, a domain of legal values is associated to each variable, to all the functions and to all the arguments of every predicate. There is a particular domain, which should be stressed: the temporal domain [8, 9, 10]. This domain is numerical in nature and it is associated with a total order and arithmetic operators. It is fixed at the specification time.

All the variables, functions and predicates are divided in time independent and time dependent ones [8, 9, 10]. All the time independent items do not change with time, but the time dependent items can be true at a time t and false at a time $t+\delta$. This division permits to represent time-varying elements in the time.

As a first-order language, TRIO includes all the typical elements of these languages, such as variables, functions, predicates and also the prepositional connectors and the quantifiers.

Besides all these elements, a TRIO formula can also be composed of

[1] In a first-order logic the set of statements of the prepositional logic is replaced by a set of predicates, functions and variables. The variables are allowed to be quantified [16].

temporal operators. The basic temporal operator of TRIO is "Dist", defined in [8, 9] in such a way that "If F is a TRIO formula and t is a term of the temporal type, then Dist(F,t) is a TRIO formula, that is satisfied at the current time instant iff F holds at the distance of t time units from the current one."

With the *Dist* operator, it is possible to derive all the other TRIO temporal operators. A list of all these operators is given below, with the definition and an explanation for each operator.

As TRIO is a first-order language, we can define in TRIO the concepts of validity and satisfiability. A TRIO formula F is satisfiable in an interpretation I, if there exists a time t such that F is true at t. In this case, I constitutes a model for F. A TRIO formula is temporally valid in an interpretation I, if Dist(F, t) forall t of Time. If a TRIO formula is valid at any moment in time or cannot be satisfied in any interpretation, the formula is said to be time invariant. A temporally closed formula is time invariant [10].

According to [10] "A TRIO formula is classically closed if all of its time independent variables are quantified, it is temporally closed if it does not contain time dependent variables or predicates, or of it has either "Som" or "Alw" as the outermost operator, or finally if it results from the prepositional composition or classical closure of temporally closed formulas." A specification of a real-time system can be defined as a TRIO formula, which is classically and temporally closed.

The following derived TRIO operators are defined in [8, 9, 10] based on the *Dist* operator as follows:

(the following ones are defined with the Dist operator)
Futr(F, d) = d $\geq$ 0 & Dist(F, d)
means that F will hold at a distance of d time units in the future
Past(F, d) = d $\geq$ 0 & Dist(F, d)
means that F has held at a distance of d time units in the past
AlwF(F) = $\forall$ d (d > 0 & Dist(F, d))
means that F will hold in all future time instants
AlwP(F) = $\forall$ d (d > 0 & Dist(F, -d))
means that F has held in all past time instants
SomF(F) = $\exists$ d (d > 0 & Dist(F, d))
means that F will hold sometimes in the future
SomF(F) = $\exists$ d (d > 0 & Dist(F, -d))
means that F has held sometimes in the past

Alw(F) = $\forall$ d (Dist(F, d))
means that F holds in every time instant of the temporal domain
Som(F) = $\exists$ d (Dist(F, d))
means that there is a time instant where F holds
Lasts(F, d) = $\forall$ d' (0 < d' < d & Dist(F. d'))
means that F will hold for the next d time units
Lasted(F, d) = $\forall$ d' (0 < d' < d & Dist(F. -d'))
means that F has held for the last d time units
Within(F, d) = $\exists$ d' (0 < d' < d & Dist(F, d'))
means that F will occur within d time units

(the following ones are defined by using the operator defined above)
UpToNow(F) = $\exists$ d (d > 0 & Past(F, d) & Lasted(F, d))
means that F held for a non null time interval that ended at the current instant
Becomes(F) = F & UpToNow(NOT F)
means that F holds at the current instant but it did not hold for a non null interval that preceded the current instant
Until(F1, F2) = $\exists$ d (d > 0 & Futr(F2, t) & Lasts(F1, t))
means that F2 will happen in the future and F1 will be true until then
Since(F1, F2) = $\exists$ d (d > 0 & Past(F2, t) & Lasted(F1, t))
means that F2 has happened in the past and F1 has been true since then
NextTime(F, d) = Futr(F, d) & Lasts(NOT F, d)
means that the first time in the future when F will hold is d time units apart from now
LastTime(F, d) = Past(F, d) & Lasted(NOT F. d)
means that the last time in the past when F has held was d time units apart from now

The following axiom, for example, defines a cycle of activations:

ActivationCycle:
Becomes(activation) $\rightarrow$
Lasts(activation, δ_1)
$\wedge$ Futr(NextTime(activation, δ_2). δ_1)

if the predicate "activation" becomes true. then it is true for δ_1 time units

and in δ_2 time units in the future, the predicate "activation" will become true for δ_1 time units the next time.

Finally, it is interesting to note that the operators of temporal logic and those of several versions of temporal logic can be defined as TRIO derived operators.

Another important feature of TRIO is that it is a specification language designed to be executable [16].

1.2 The TRIO+ Language

TRIO is a language for specifying "in the small" [8], but there is a need, to specify also more complex systems, i.e. to write specifications "in the large". TRIO+ is the adaptation of TRIO to the specification "in the large" by using Object Oriented concepts.

1.2.1 Simple classes

A simple class regroups TRIO axioms and the corresponding declarations [8, 10].

In classical object-oriented languages the instantiation of a class is in some way an abstraction of a memory cell, i.e. an information container whose contents obey the law class. As TRIO+ is a pure logic language, its variables must be intended in a purely mathematical sense. An instance of a class is a model for axioms of the class.

An example of a simple class is the following specification of a detector.

Class DETECTOR [8]
Visible activation, position, quantum
Temporal Domain real
TI Items
consts $\delta, \delta_1, \delta_2$: real;
each of this constants represents a certain number of time units
TD Items
predicates activation, quantum, acquisition
variables position: n: {open, closed}

Axioms

ActivationCycle:

Becomes(activation) $\rightarrow$
Lasts(activation. δ_1)
$\wedge$ Futr(NextTime(activation, δ_2), δ_1)

if the predicate "activation" becomes true. then it is true for δ_1 time units and in δ_2 time units in the future, the predicate "activation" will become true for δ_1 time units the next time.

AcquisitionDelay:

Becomes(activation) $\leftrightarrow$ Futr(acquisition, δ)

if the predicate "activation" becomes true. then the predicate "acquisition" will be true after a delay of δ time units

QuantumDetection:

quantum $\leftrightarrow$
acquisition $\wedge$ position $\neq$ Past(position, $\delta_1 + \delta_2$)

a quantum is registered if from one activation to another a transition of the position is detected

end DETECTOR

The first keyword is *class*, followed by the name of the class and both of them represent the header of the class. The header is followed by the keyword *visible*. This clause regroups all the variables, functions and predicates which can be used (seen) by other classes. All these variables, functions and predicates, in the visible clause, constitute the class interface. A predicate is a function whose result is of the boolean type.

The declarations are divided into two groups, *TD_Items* (Time Dependent Items) and *TI_Items* (Time Independent Items). The keyword TD_Items is followed by all the time dependent and the keyword TI_Items is followed by all the time independent variables, functions and predicates. Finally, all the *axioms* are TRIO formulas. The universal quantifier and the Alw temporal operator are implicit for all the axioms.

It is possible to give a name to an axiom, so it can easily be referred to. The name precedes the axiom and is followed by a colon. The possibility of informal comments exists as well.

The class Detector can be represented graphically as shown in figure 1.1.

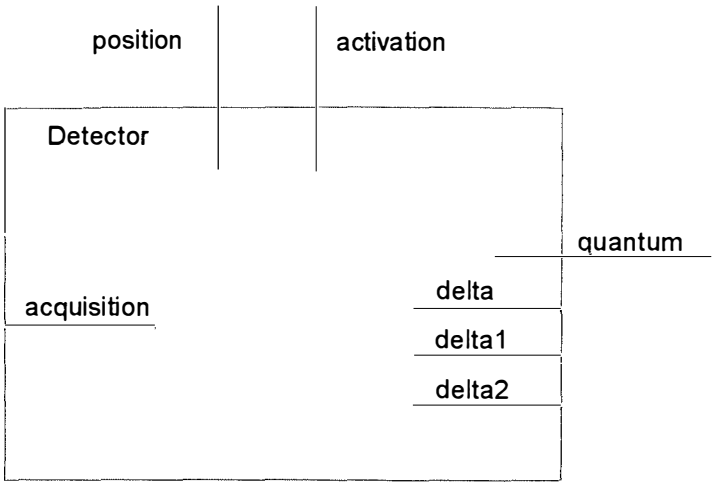

Figure 1.1: The graphic representation of the class Detector

The class is represented as a box with its name written at the top left. Each name of an item is written on a separate line. This line is internal for all items. If the item is visible then the correspondent line continues outside the box.

1.2.2 Structured classes

A class, composed of parts belonging to other classes, which are called modules, is a structured class [8, 10]. In this way, TRIO+ modular specifications can be built. This method is especially interesting, since this way the axioms contained in a rather complex class can easily be regrouped into different parts. Each of these parts can be represented by a module. The *connections* define the flow of information between the class and its modules and between the modules.

For example, the Energy_Meter contains three modules: a class disc, detector and totalizer. The specification of this Energy_Meter is represented as follows:

Class ENERGYMETER [8]
Visible measurement, ϕ
TD Items
vars measurement: integer,
ϕ: real;
Modules disc: DISC;
detector: DETECTOR
totalizer: TOTALIZER
Connections
{ (ϕ disc.ϕ)
(disc.activation detector.activation)
(disc.position detector.position)
(detector.quantum totalizer.quantum)
(totalizer.measurement measurement) }
Axioms
measureIncrease:
detector.quantum $\rightarrow$
totalizer.measurement
$<$ Futr(totalizer.measurement, 0.1)
the detection of a quantum by the detector has as effect that the measurement is done by the totalizer and this measurement is inferior to the measurement which will be done in 0.1 time units in the future
end ENERGYMETER

The graphic representation of class EnergyMeter is shown in figure 1.2. In a structured class, each line represents a connection.

Unfortunately, recurive definition of a class is not possible. A module, on its turn, can also be a structured class.

Further, the temporal domain of the modules must be the same as that of the structured class that includes it.

A module, however, is not a logical symbol. So, for example, if a class has in its visible clause the whole interface of one of its modules, all the items of the modules interface must be listed in the visible clause of the class. It is not possible to list a module in a clause in the same way then variables, functions or predicates.

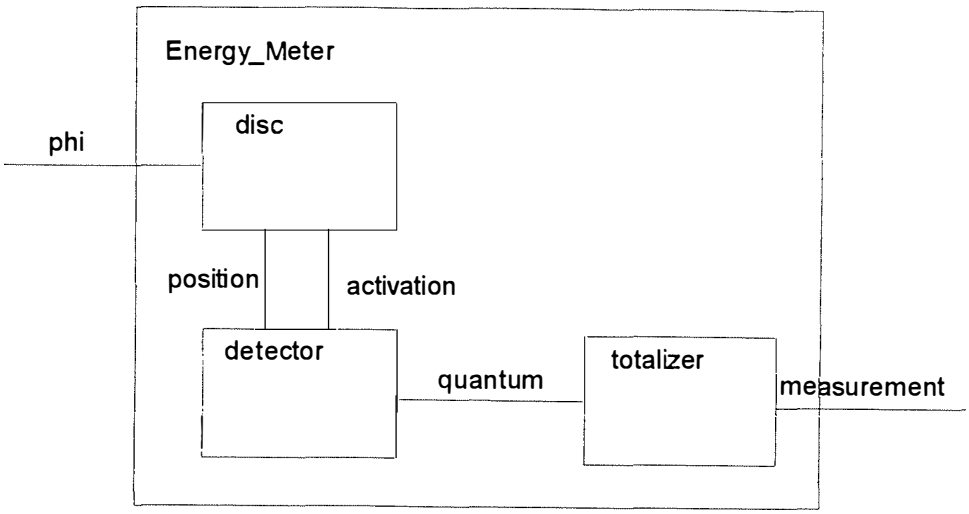

Figure 1.2: The graphic representation of the class EnergyMeter

1.2.3 Genericity

TRIO+ supports a genericity mechanism [8, 10]. A generic class has one or more parameters, that can represent any type. These parameters are listed behind the name of the class between brackets. Only the interface of modules have to be defined, with a clause of the kind:

where class-parameter has list_of_the_elements_of_the_interface.

If a generic class is instantiated with a class, having the same number of parameters and the defined interface of modules correspond, then the instantiation gives a non-generic class. This instantiation is done by a clause of the kind:

class class-name is generic_class_name.

In the following example [8], an Energy_Meter is defined with a GenericDisc. The composition of the Energy_Meter is represented graphically in figure 1.2.

A specification of an EnergyMeter without reference to a particular choice of a Disc is obtained by defining a generic class:

Class PARAMETRICENERGYMETER[GENERICDISC]
where genericDisc **has** activation, position, ϕ
... **Modules** disc: GENERICDISC;
detector: DETECTOR
totalizer: TOTALIZER
... **end** ENERGYMETER

The specification of a particular EnergyMeter is obtained by instantiating the formal parameter genericDisc with the actual parameter. In our example shown in figure 1.3, the disc is equipped with three photocells, which is a particular choice.

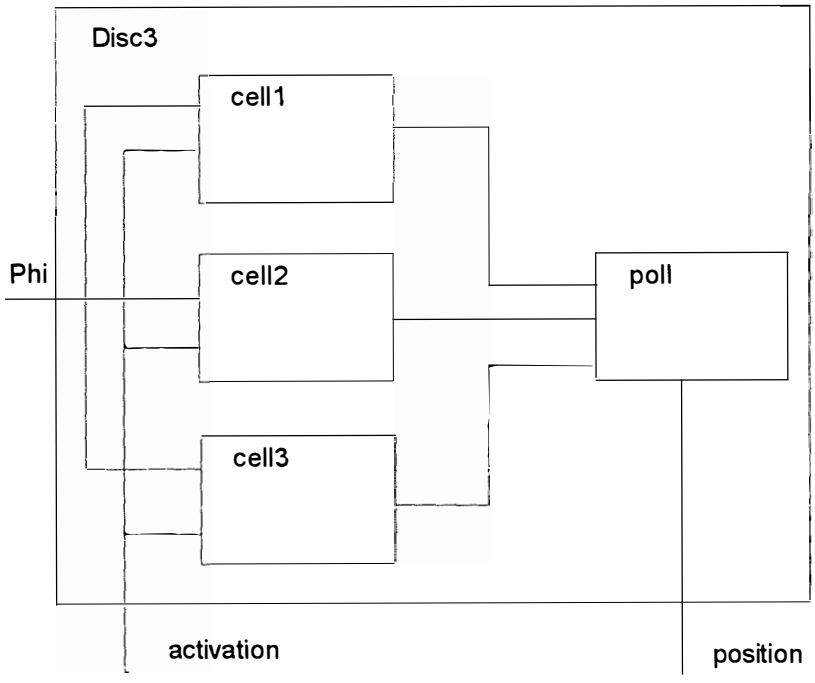

Figure 1.3: A disc with triple-photocell

In this case, the actual parameter for GenericDisc is Disc3:

Class EnergyMeter3 **is** ParametricEnergyMeter[Disc3]

1.2.4 Inheritance

Inheritance is another OO-mechanism, supported by TRIO+ [8, 10]. This mechanism allows a class, called subclass, to receive attributes from another class, called superclass. This is done by adding an *inherit* clause after the class-header. The keyword inherit is followed by the name of the superclass. In a subclass, axioms may be redefined freely. In this way, a subclass can admit an entire different semantic than its superclass.

The mechanism of inheritance is subject to a few constraints:

- As TRIO+ ensures the monotonicity of external interface, the interface of a subclass must be the same as the interface of the superclass.
- Items, modules and axioms may be added to a subclass, but attributes cannot be removed from the subclass.
- The redefinition can only use a subclass of original module class.
- All the attributes, which are redefined in the subclass, must be listed in the *redefine* clause, following the inherit clause.
- The directed graph defined by inheritance relation must be acyclic.

Besides the mechanism of simple inheritance, multiple inheritance is possible too. All, what was said above about the simple inheritance holds for multiple inheritance, and in addition, there are two more constraints:

- In the case of multiple inheritance, name clash may occur in the case when two or more attributes of the same name are inherited from different superclasses. A name clash can be solved by renaming the attributes in a way, that name clash does no more occur.
- A class cannot inherit from two different classes, if these two classes inherit their attributes from the same superclass and if at least one has redefined, renamed or instantiated an attribute.

The following class StableDetector is defined in inheriting from the class Detector and in redefining the axiom quantumDetection.

Class STABLEDETECTOR [S]
inherits Detector [**redefine** quantumDetection]
Axioms

quantumDetection:

$$\text{quantum} \leftrightarrow \text{acquisition} \wedge \left\{ \begin{array}{c} \text{position} = \text{Past}(\text{position}, \delta_1 + \delta_2) \\ \wedge \text{position} \neq \text{Past}(\text{position}, 2(\delta_1 + \delta_2)) \\ \wedge \text{Past position}, 2(\delta_1 + \delta_2)) \\ \text{Past}(\text{position}, 3(\delta_1 + \delta_2)) \end{array} \right\}$$

a quantum is registered only when two consecutive similar position values are detected

end ENERGYMETER

Chapter 2

Introduction to PVS

PVS stands for "Prototype Verification System" and as the name suggests, it is a prototype for a system for specification and verification based on higher-order logic [1] with a very rich type system developed at the Computer Science Laboratory of SRI International [12, 13, 14, 16].

PVS attempts to provide a support to write correct specification and to confirm it. To do this, it offers a combination of an expressive specification language and a theorem prover which is interactive and highly mechanized.

In this chapter we first describe briefly the PVS Specification Language and the PVS Proof Checker before we treat the example of the Airline Reservation System.

2.1 The PVS Specification Language

As mentioned before, PVS is a simply typed higher-order language.

The type system of PVS allows an early detection of errors by a rigorous type checking. The PVS Typechecker generates type correctness conditions (TCCs) when the user must show that the proof obligations are correct. For instance, the predicate subtype makes it possible to define functions, that are only partially defined, as total function. As a consequence a predicate subtype is therefore undecidable and the PVS Typechecker generates a TCC, which must be proven to show that instances of the subtype satisfy the

[1] An higher-order Logic is an extension of the first-order Logic. It adds to the first-order logic the possibility to quantify predicates and functions. and it allows a function to take other functions as arguments [16].

predicate.

A PVS specification is devided into various theories, this division allows the user to structure the specification. A Theory is composed of type and variable declarations. The declared *axioms* are considered as assumptions and are not checked. All the other statements, as *lemmas* and *theorems*, have to be proved.

It is possible to import a theory into another one by the *importing* clause, this has as effect that names declared in one theory can be shared between different theories. A theory can not import, directly or undirectly, itself. This means that the importing chain must form a directed acyclic graph.

2.2 The PVS Proof Checker

The PVS Proof Checker is based on the Gentzen's sequent semantic and is supported by arithmetical and logical decision procedures [4, 16].

A sequent in the Gentzen's-sequent semantic is an ordered pair (Γ, Δ) of sets of formulas $(\Gamma \vdash \Delta)$ [16]. The formulas in Γ are called the antecedent formulas and those in Δ are called consequent formulas. An inference rule is composed of upper sequents called premises and a lower sequent called conclusion. An axiom is only composed of a conclusion, it has an empty set of premises.

A proof is a tree of sequents. Each sequent is generated from its parent sequent by an inference rule. The proofs are done backwards, this means that the initial proof obligation is split into smaller obligations and in this way all the branches and leaves of the proof tree are generated. A proof is completed when there are no remaining unproved leaf sequents in the proof tree, so that all the leaves are trivially true [14].

In same way in PVS, a proof is carried out as a dialogue between the user and the Proof Checker. The user tells the proof checker the next step to perform and the proof checker applies it by breaking the goal progressively into simpler subgoals or obvious truths or falsehoods. The Proof Checker plays also the skeptical part in such a dialogue, which rejects any argument that is not consistent.

The PVS Proof Checker also offers the possibility to include new decision procedures e.g., those which are specific to an encoded logic, by defining new strategies. A strategy allows the user to apply proof rules without understanding the semantic details of the proof steps and without having to

guide the Proof Checker explicitly through it [16].

2.3 The Example of the Airline Reservation System

To familiarize with the techniques and concepts of PVS, we carried out some proofs with the reference to the 'Airline Reservation System' case study.

This case study is an automated airline seat assignment system [3], the informal requirements of the system are stated as follows. The system should make seat assignments for passengers on scheduled airline flights and maintain a database of the seat assignments. Different aircraft types should be supported. It should also be possible to the passenger to specify preferences for a seat type e.g., window or aisle. The operations of making and canceling seat assignments should be provided by this system.

```
basic_defs: THEORY
BEGIN
    nrows: posnat
    nposits: posnat

    row: TYPE = {n: posnat|1<=n & n<=nrows} CONTAINING 1
    position: TYPE = {n: posnat|1<=n & n<=nposits} CONTAINING 1
```

The CONTAINING clause ensures "the existence of a member of the user-defined subtype" [3].

```
    flight: TYPE
    plane: TYPE
    preference: TYPE
    passenger: TYPE

    seat_assignment: TYPE = [#seat: [row,position],
                             pass: passenger#]
```

'Seat_assignment' is defined in using a record constructor and each record contains a passenger identification and the assigned seat. The type of a seat is the 2-tuple of row and position, which defines one single seat on a flight.

```
flight_assignments: TYPE = set[seat_assignment]
```

'Flight_assignment' represents the entire set of seat assignments for a flight.

```
assn_state: TYPE = function[flight -> flight_assignments]
```

'Assn_state' stands for a complete flight-reservation database, which consists of the mapping of a flight identifier into the flight's current set of seat assignments.

```
flt: VAR flight

initial_state: function
    [flight -> flight_assignments] =
    (LAMBDA flt: emptyset[seat_assignment])
```

LAMBDA is a syntactic key word and "means the following text up to the colon are the formal arguments for this function" [3].
Initially, each flight has no assignments, which is stated by the function 'Initial_state'.

```
seat_exists: function
    [plane,[row,position] -> bool]
```

The 'Seat_exists' function is true when the given seat exists physically on the indicated plane. This function remains uninterpreted because the specification is not restricted to a particular plane type.

```
meets_pref: function
    [plane,[row,position],preference -> bool]
```

The 'Meets_pref' function is true if there exists a seat on the plane which matches with the preference of the passenger.

```
    aircraft: function
        [flight -> plane]
```

The type of the airplane assigned to a particular flight is given by the 'Aircraft' function.

```
END basic_defs

ops: THEORY
BEGIN
    IMPORTING basic_defs

    flt: VAR flight
    pas: VAR passenger
    as, s1: VAR assn_state
    a,b,x: VAR seat_assignment
    pref: VAR preference
    seat: VAR [row, position]

    cancel_assn: function
        [flight, passenger, assn_state -> assn_state] =
        (LAMBDA flt, pas, s1:
            s1 WITH [(flt) := {a|member(a,s1(flt))
            & pass(a) \= pas}])
```

The 'Cancel_assn' function has as effect to remove all seat assignments of the passenger 'pas' for flight 'flt'.

```
    pref_filled: function
        [assn_state, flight, preference -> bool] =
        (LAMBDA as, flt, pref:
        (FORALL seat: meets_pref(aircraft(flt), seat, pref)
        IMPLIES (EXISTS a: member(a, as(flt))
                                & seat(a) = seat)))
```

The 'Pre_filled' function is true, when there is no seat available that meets the passenger's preference. The 'Meets_pref' function states all the seats that meet the passenger's preference.

```
next_seat:function
    [assn_state, flight, preference -> [row, position]]
```

The 'Next_seat' function selects the next seat matching with the passenger's preference, attributed to the passenger.

As the selection algorithm is unspecified, e.g. the function is left uninterpreted, a general property, expressed in the 'Next_seat_ax' axiom, is needed for one of our proofs.

```
next_seat_ax: AXIOM
    NOT pref_filled(s1, flt, pref)
    IMPLIES seat_exists(aircraft(flt), next_seat(s1, flt, pref))

pass_on_flight: function
    [passenger, flight, assn_state -> bool] =
    (LAMBDA pas, flt, s1: (EXISTS a: pass(a) = pas
        & member(a, s1(flt))))
```

The function 'Pass_on_flight' is true, if the passenger already has a seat on the plane.

```
make_assn: function
    [flight, passenger, preference, assn_state
       -> assn_state] =
    (LAMBDA flt, pas, pref, s1:
    IF pref_filled(s1, flt, pref)
       OR pass_on_flight(pas, flt, s1)
    THEN s1
    ELSE (LET a=(#seat := next_seat(s1, flt, pref),
            pass := pas#) IN s1
         WITH[(flt) := add(a, s1(flt))])
    ENDIF)
```

The 'Make_assn' function performs a seat assignment for a passenger 'pas' on flight 'flt' if:

- *there is a seat available that meets the passenger's preference*
- *the passenger has no seat on this plane yet.*

The following functions are stating the invariant of the system.

```
% Invariants

    existence: function
        [assn_state -> bool] =
        (LAMBDA as: (FORALL a, flt: member(a, as(flt))
        IMPLIES seat_exists(aircraft(flt),seat(a))))
```

The 'Existence' function ensures that no assignment of nonexisting seats is made to passengers.

```
    uniqueness: function
        [assn_state -> bool] =
        (LAMBDA as: (FORALL a, b, flt:
        member(a, as(flt)) & member(b, as(flt))
        & pass(a) = pass(b)
        IMPLIES a = b))
```

The 'Uniqueness' function makes sure that no multiple seat assignments are made to a single passenger.

```
    assn_invariant: function
        [assn_state -> bool] =
        (LAMBDA as: existence(as) & uniqueness(as)
                    & one_per_seat(as))
```

The complete invariant of this system is stated with the 'Assn_invariant' function.

```
    cancel_assn_inv: THEOREM
        assn_invariant(s1)
        IMPLIES assn_invariant(cancel_assn(flt, pas, s1))
```

The 'Cancel_assn_inv' Theorem guaranteers that after a 'Cancel_assn', the invariant is still respected.

```
MAe: THEOREM
    existence(s1)
    IMPLIES existence (make_assn(flt, pas, pref, s1))
```

The 'MAe' Theorem ensures, that after a 'Make_assn' no assignment attributes a nonexisting seat to a passenger.

```
MAu: THEOREM
    uniqueness(s1)
    IMPLIES uniqueness(make_assn(flt, pas, pref, s1))
```

The 'MAu' Theorem guaranteers, that after a 'Make_assn', no multiple seat assignment to a single passenger exists.

```
make_assn_inv: THEOREM
    assn_invariant(s1)
    => assn_invariant(make_assn(flt, pas, pref, s1))
```

The 'Make_assn_inv' Theorem ensures that after a 'Cancel_assn', the invariant is still respected.

```
make_cancel: THEOREM
    NOT pass_on_flight(pas, flt, s1) =>
    cancel_assn(flt, pas, make_assn(flt, pas, pref, s1))
              = s1
```

The 'Make_cancel' Theorem states, that a 'Cancel_assn' will cancel a 'Make_assn' operation.

```
END ops
```

The system specification originally, as reported before, includes axioms avoiding the two following anomalies of the system state:

Assigning nonexistent seats to passengers
Assigning multiple seats to a single passenger.

These are respectively guaranteed by the following functions:

```
existence: function
    [assn_state -> bool] =
    (LAMBDA as: (FORALL a, flt:
    member(a, as(flt)) IMPLIES seat_exists(aircraft(flt),
                                           seat(a))))

uniqueness: function
    [assn_state -> bool] =
    (LAMBDA as: (FORALL a, b, flt:
    member(a, as(flt)) & member(b, as(flt))
    & pass(a) = pass(b)
    IMPLIES a = b))
```

We also added axioms ruling out the following one:

Assigning more than one passenger to a single seat.

Now, we explain in detail how we have introduced the third invariant in the system. First of all, we have added a function which defines the third invariant, as follows:

```
one_per_seat: FUNCTION
    [assn_state -> bool] = (LAMBDA as: (FORALL a, b, flt:
    member(a, as(flt)) & member(b, t)) & seat(a) = seat(b)
    IMPLIES a = b))

next_seat_ax_2: AXIOM
    (FORALL a: member(a, s1(flt))
    IMPLIES seat(a) /= next_seat(s1, flt, pref))
```

The previous axiom states that a seat attributed to a new passenger in a flight 'flt' is different from the seat already attributed to the passengers of flt. This axiom is crucial for one of our proofs because the 'Next_seat' function is left uninterpreted.

The following theorems proves, that this new axiom is valid in the system and that all the other axioms are still valid.

```
cancel_assn_inv: THEOREM
    assn_invariant(s1)
    IMPLIES assn_invariant(cancel_assn(flt, pas, s1))

make_assn_inv: THEOREM
    assn_invariant(s1)
    => assn_invariant(make_assn(flt, pas, pref, s1))
```

These two theorems state respectively that 'cancel_assn' and 'make_assn' maintain the invariant. This invariant ('assn_invariant') ensures that in every moment of the system:

- every assigned seat to a passenger really exists on the corresponding flight
- only one seat is assigned to a passenger on one flight
- only one passenger is assigned to a seat on one flight.

```
assn_invariant: FUNCTION
    [assn_state -> bool] =
    (LAMBDA as: existence(as) &
    uniqueness(as) & one_per_seat(as))
```

The second theorem is proved by using 'MAe', 'MAu' (two theorems which are already proved in the system, guaranteeing the 'existence' and 'uniqueness' invariant of seat assignments) and 'MAs', the new theorem which states that after a new assignment, the 'one_per_seat' invariant is still valid.

```
MAs: THEOREM
    one_per_seat(s1)
    IMPLIES one_per_seat(make_assn(flt, pas, pref, s1))

initial_state_inv: THEOREM
    assn_invariant(initial_state)
```

This theorem shows that the initial state of the system satisfies the invariant.

The following two theorems are stating in particular that the 'one_per_seat' invariant is always respected in the system.

The first one guaranteers that the 'one_per_seat' invariant stays valid after canceling an assignment ('cancel_inv_one_per_seat') which was already proved by the fact that 'cancel_assn' has no effect on the global invariant ('existence', 'uniqueness' and 'one_per_seat' invariant).

'Initial_one_per_seat' shows that the 'one_per_seat' invariant is satisfied in the initial state which of course was already done by proving 'initial_state_inv' for the global invariant.

```
cancel_inv_one_per_seat: THEOREM
    one_per_seat(s1)
    IMPLIES one_per_seat(cancel_assn(flt, pas, s1))

inital_one_per_seat: THEOREM
    one_per_seat(initial_state)
```

By proving that the global invariant and in particular the 'one_per_seat' invariant holds in the initial state and is preserved after a 'make_assn' and/or a 'cancel_assn', it is proved that the system always preserves the invariant.

Putative theorems are used to confirm our understanding of the specified system.

The 'make_putative' theorem states that if a seat that matches the specified preference, is available then there exists an assignment which attributes a seat to this passenger.

```
make_putative: THEOREM
    NOT pref_filled(s1, flt, pref) =>
    (EXISTS(x: seat_assignment):
        member(x, make_assn(flt, pas, pref, s1)(flt))
        & pass(x) = pas)
```

The 'cancel_putative' theorem states that after a 'cancel_assn' no more assignment exists which attributes a seat to this passenger.

```
cancel_putative: THEOREM
    NOT (EXISTS (a: seat_assignment):
    member(a, cancel_assn(flt, pas, s1)(flt))
    & pass(a) = pas)
```

The different steps of the proofs of each theorem can be found in the appendix A.

Chapter 3

The Encoding of TRIO in PVS

3.1 Introduction

As explained in the previous chapter, PVS is an higher-order logic that does not include the time notion, while TRIO is a temporal first-order logic.

The goal of the encoding of TRIO in PVS is to use the proof checker of PVS to prove properties expressed in TRIO [1, 17]. (Further on in the text, the term TRIO/PVS will be used to refer to the encoding of TRIO in PVS.) TRIO/PVS contains all the necessary definitions and axioms to express TRIO formulas and their properties in PVS. Strategies have been developed, so that the proof checker of PVS is able to support automatic execution of a few demonstration steps.

A semantic encoding of TRIO/PVS seems more appropriate to the envisaged goal than a syntactic encoding, even if it presents two disadvantages. An syntactic encoding would not provide "the high level of automated support through decision procedures that would be available in the base logic of a semantic encoding" [7]. "A semantic encoding involves giving meaning to each construct of the source logic (TRIO) by definitions in the base logic (PVS)" [7]. The first inconvenient is that the encoded formulas may be considerably modified compared to the original source language. The second one is that the user must revert to the basic logic during the construction of proofs, even if the proof rules of the source logic are encoded.

The encoding of TRIO in PVS has been done in such a way, that all the necessary definitions and axioms are hidden to the user of TRIO/PVS. In some way, the user has the impression, that the proof checker of PVS is a

TRIO tool.

A TRIO formula remains uninterpreted in PVS. But a semantic function called 'now' assigns a boolean value to a TRIO formula with respect to the current time instant. This function is also uninterpreted because PVS only knows the signature of this function, i.e. "we do not give a description a typed based on the other ones predefined in PVS" [17]. Nevertheless, two assumptions are made on an uninterpreted type in PVS, first of all this type is not empty and it has to be disjoint from all the other types.

To introduce the notion of time in PVS and avoiding to do it explicitly, the following solution was adopted [17]. Two different kinds of axioms are introduced:

- those which describe the fundamental properties of the *Dist* operator and basically consists of the two following axioms. The first axiom ("dist_dist") states that: If a holds in d_2 time units from now and if this is the case d_1 time units from now. then a holds in $d_1 + d_2$ time units from now. The second one ("dist_zero") states that: If a holds 0 time units from now, then a is true now.

```
dist_dist: AXIOM
    Dist(Dist(a, d2), d1) = Dist(a, d1+d2)

dist_zero: AXIOM
    Dist(a, 0) = a
```

- those which express the interpretation of any other operator translated through *Dist* at the outermost level. For example in the "dist_and_exp" axiom which is reported below, the & operator is interpreted. If a and b hold in d time units from now, then a holds in d time units from now and b holds in d time units from now.

```
dist_and_exp: AXIOM
    now(Dist(a & b, d)) =
    (now(Dist(a, d)) & now(Dist(b, d)))
```

With this axiom and the previous ones, it is possible to establish the following theorem:

```
and_exp: THEOREM
    now(a & b) = (now(a) & now(b))
```

and to perform all the necessary transformations for the interpretations and strategies of PVS.

In TRIO/PVS there are two different types of formulas, time independent (TI_Formula) and time dependent ones (TD_Formula). So, all the TRIO operators have to be distinguished, whether their arguments are time independent or time dependent. For the same reason, the encoding needs to introduce time dependent quantifiers.

3.2 Translating a TRIO+ specification into TRIO/PVS

In order to be able to use TRIO/PVS to prove formal specification written in TRIO+, this specification must be transformed to a PVS like specification.

In the following section, we prove some properties of the Energy_Meter that are ensured by its specification. Before explaining the different proofs, the major steps of the translating from the TRIO+ specification of the Energy_Meter example [11] to TRIO/PVS are explained.

1. As PVS does not know the formal specification of TRIO, TRIO/PVS was used by `importing trio[int]` in the first Theory. While the TRIO/PVS is imported, the type of Time is also defined for all the theories of this specification. In `importing trio[int]`, the type of Time is declared as being an integer([int]).

2. So for each TRIO+ class, a Theory in PVS was created. First of all, we can stay much closer to the formal specification in maintaining the same division and we reduce the complexity of the TRIO/PVS document in dividing the whole theory into various smaller Parts. They are all linked together through the IMPORTING clause, which establishes a sort of hierarchy. As we explain above, the type of Time must be the same for each of this theories.

3. In a TRIO+ class we can find different parts:

   (a) In the declaration of *Visible* in TRIO, we find all the variables, functions and predicates, which can be used by other classes. The constants, functions and predicates, which are represented in a TRIO visible clause, are only declared in one PVS theory. In all the other PVS theories, which have imported directly or indirectly the theory with the declaration, can simply reuse them. Actually, only the variables have to be redeclared if they are reused in another theory. In this way, the visibility of the constants, functions and predicates is maintained.

   (b) All the *Time Independent Items* are declared as such in PVS without any further problems, because all the declarations in PVS are time independent.

   (c) For the *Time Dependent Items* we have to introduce a mechanism to consider all the declared items as time dependent. As the declarations in PVS are time independent, these variables are declared as time independent and a predicate for the time dependent variables is created in the following way:

```
- IMPORTING TD_Var(Time, T)
  with Time is the type of time,
     T is a type of a variable t.
  The type of time is an NONEMPTY_TYPE from INT.

- Predicate : TD_Var(Time, T).TD_var
  where Predicate is the name of the predicate and
      TD_Var(Time, T).TD_var is of type BOOLEAN.
```

   e.g. the declaration of the time dependent predicate "position":

```
IMPORTING TD_Var[Time Pos]
position: TD_Var[Time, Pos].TD_var
```

   The value of such a predicate, applied to a time independent variable, is only true for one value of the time independent variable at a given time t of type T.

   (d) All time dependent predicates have been declared as TD_Formulas. A TD_Formula is the type of time dependent predicates, which is defined in TRIO/PVS.

(e) All the axioms have to be in a trio-basic form, this means that the outermost operator is a TRIO temporal operator. To do this, we put all the axioms of the formal TRIO+ specification into an *Alw* predicate. This does not change the formal specification, because the *"Alw* predicate is always implicit in TRIO+.

3.3 Personal Contribution to TRIO/PVS

(a) The *Since* operator was not yet included into TRIO/PVS until now.

As explained above, all the operators can be expressed in an axiom using the *Dist* operator and in particular the "Since" operator. We use the "Past" and "Lasted" operators to define the "Since" operator, because they allow us to express the *Since* operator more easily. The "dist_since_exp" axiom, reported below, allows us to establish the "Since_exp" theorem. It is possible with both of them to add the *Since* operator in the strategies without any problem.

All the other theories and strategies, *Since* is treated as a composition of the operators "past" and "lasted", which are predefined. In proofs of theorems containing the *since* operator, we have to flatten or split the expression with the "since" operator and then continue the strategy followed to prove the theorem.

The *Since* operator is added to TRIO/PVS as follows:

```
- in trio_temp.pvs

  Since: (TD_Formula, TD_Formula -> TD_Formula)

  dist_since_exp: AXIOM
      now(Dist(Since(a, b), d2)) =
      now(Dist(Past(, d) & Lasted(a, d), d2))

  since_exp: THEOREM
      now(Since(a, )) =
      now(Past(, d) & Lasted(a, d))
```

```
- in temp_ext add

  Since: (TD_Formula, TI_Formula -> TD_Formula)
  Since: (TI_Formula, TD_Formula -> TD_Formula)
```

The *Since* operator has also to be added to the strategies.

As *Since* is defined as a composition of the *Past* and *Lasted* operators, we only need to add the way of decomposing the *Since* operator to the strategies.

This is done, as follows, in applying to the sequent with the *Since* operator, target the axiom which is defined above. Further, in the single-flat strategy, we apply 'flatten', a disjunctive simplification [1] to the target. In the single-split strategy 'split', a conjunctive splitting [2] is applied to the target. As this is the only way to decompose *Since*, we have only one 'try' clause. If this one fails, the simplification is not possible and 'skip' is applied to continue with the strategy.

```
- in single-flat add

  ((eq op3 '|Since|)
    ' (try (rewrite "dist_since_exp", target)
           (flatten, target)
           (skip))
      )

  ((eq op2 '|Since|)
    ' (try (rewrite "since_exp", target)
           (flatten, target)
           (skip))
      )
```

[1] "Disjunctive simplification transforms each indicated formula into a list of formulas that contains no disjuncts by repeatedly transforming" [14].

[2] "A conjunctive formula A in a goal sequent of the form $\Gamma, A \vdash \Delta$ or $\Gamma \vdash A, \Delta$ is split by collecting lists of antecedent and consequent formulas by recursively collecting subformulas of A [14].

```
- in single-split add

  ((eq op3 '|Since|)
    ' (try (rewrite "dist_since_exp", target)
           (split, target)
           (skip))
      )

  ((eq op2 '|Since|)
    ' (try (rewrite "since_exp", target)
           (split, target)
           (skip))
      )
```

(b) PVS does not know the operator "mod". This operator (x mod y) gives the rest of the division of x by y. This means that it :

$$\exists n : ny + rest = x.$$

In our specification, the operator mod is only used in inequalities of the kind:

$$a < x \bmod y < b.$$

So, we transformed them in the following way:

$$\exists n : ny + a < x < ny + b.$$

Chapter 4

The TRIO+ Specification of the Energy_Meter

4.1 Introduction to the Energy_Meter Case Study

In the following case study, we consider a part of the specification of a digital energy and power meter. The entire specification can be found in the appendixe B. The explanation of the 'Energy_Meter' case study is mainly based on [9] and the figures are extracted from [11].

The Energy_Meter is composed of a magnetic transducer that converts the energy flow through the line into a disc rotation. In the peripheral part of the disc, transparent and opaque parts are evenly alternated, with the purpose of permitting the detection of the disc motion and its velocity by means of a photocell. In the figure 4.1 we can see the disc and the two parts of the photocell (TX and RX). The doted line represents the light signal of the photocell. If the receiver of the photocells can detect the light signal during an activation of the photocell, the part of the disc in front of the photocell is a transparent part; otherwise if the receiver can not receive the light signal during an activation, the part of the disc in front of the photocell is an opaque part. The motion and the velocity are respectively proportional to energy and power consumption.

To minimize the wear of the photocell, it is only activated during a small fraction of the total working time of the Energy_Meter. Its activation is performed according to the diagram of figure 4.2.

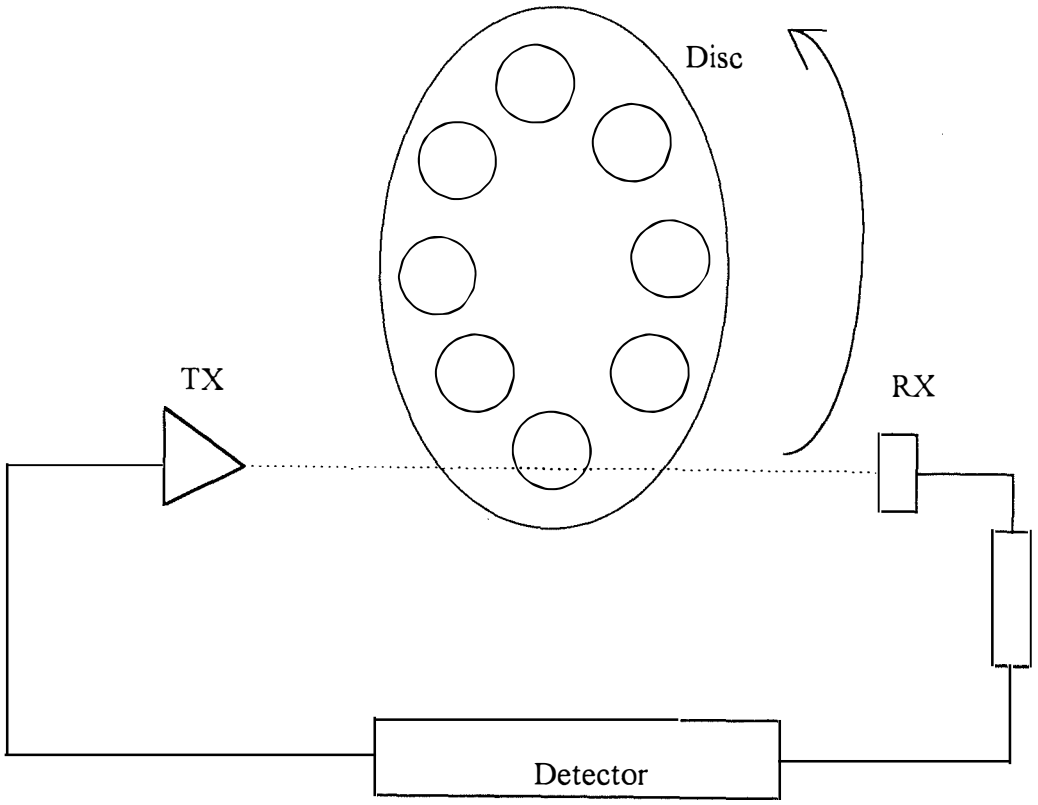

Figure 4.1: The architecture of an Optic_Disc

δ is the time the signal needs to reach a stable state
δ_1 is the time of the activation of the photocell
δ_2 is the time between two activations

Once the photocell is activated, the acquisition of its signal must be postponed by a delay of δ time units, in order to permit the signal to reach a stable state. The cell activation lasts only δ_1 time units, and it is repeated after δ_2 time units.

In this chapter, we propose a specification of an Energy_Meter with one photocell. This Energy_Meter, as it is proved in chapter five, in certain circumstances counts quantums which is only due to an external cause. Afterwards, is proposed, a similar system consisting of an Energy_Meter with two photocells, which counts only 'real' quantums. This statement is also be proved in chapter five.

At the moment of acquisition, the photocell is in a particular position relative to the disc, called α. We consider α mod 2γ, with γ the length of

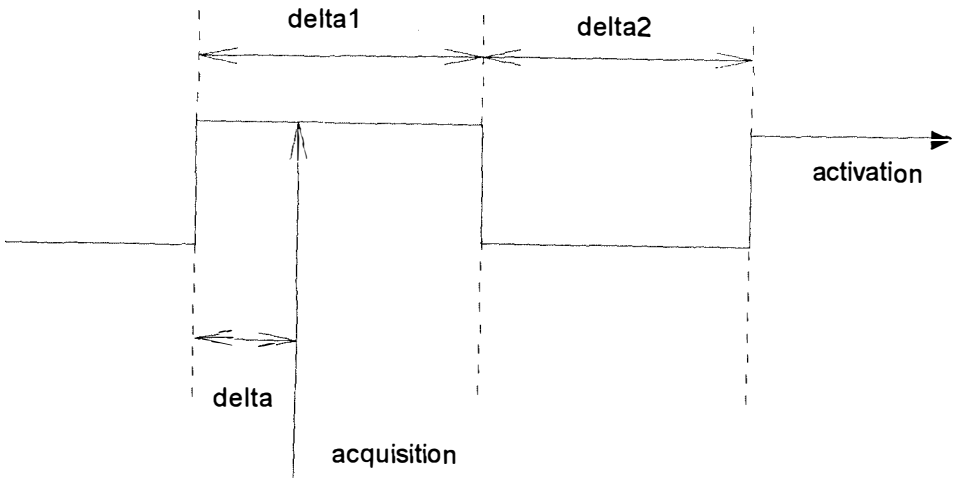

Figure 4.2: The method of sampling

one sector in order to make sure that the value of the position is in the range $[0, 2\gamma]$.

If $\frac{\delta}{2} < \alpha \bmod 2\gamma < \gamma - \frac{\delta}{2}$ then the photocell is in front of an opaque sector (ZoneF).

If $\gamma + \frac{\delta}{2} < \alpha \bmod 2\gamma < 2\gamma - \frac{\delta}{2}$ then the photocell is in front of a transparent sector (ZoneT).

If $\alpha \bmod 2\gamma \leq \frac{\delta}{2}$ OR $\gamma - \frac{\delta}{2} \leq \alpha \bmod 2\gamma \leq \gamma + \frac{\delta}{2}$ OR $2\gamma - \frac{\delta}{2} \leq \alpha \bmod 2\gamma$ then it is not sure whether the photocell is in front of an opaque sector or a transparent one. In this particular situation, we consider that the photocell is in front of a zone of indecision (ZoneI).

In figure 4.3, the various zones are represented graphically. As, the second Energy_Meter has two photocells, we have to distinguish whether it is photocell1 or photocell2 which is in front of a particular Zone. ZoneI1, ZoneF1 and ZoneV1 are the different zones in front of which photocell1 can be; ZoneI2, ZoneF2 and ZoneV2 are those in front of which photocell2 can be.

The consumption of energy is detected when the disc moves from a transparent sector to an opaque one or from an opaque one to a transparent one.

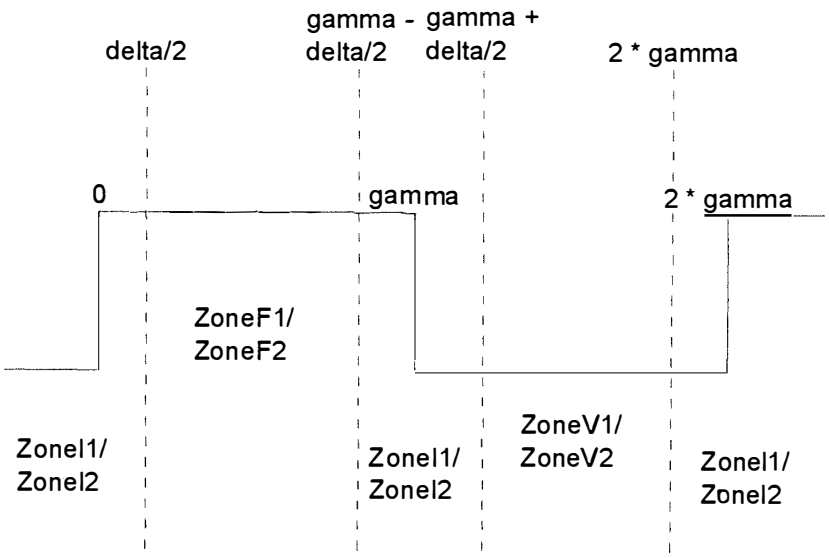

Figure 4.3: The various zones

4.2 Specification in TRIO+

A formal specification of the case study, which is described informally in the previous section is proposed in TRIO+ [11]. As seen before, in TRIO+ the classes are used to do modular specifications. In our case study the whole system is represented by the Class "Energy_Meter_I", which is composed of several modules, as shown in figure 4.4.

The module "g_ferraris" contains the technical information of the particular architecture of the Energy_Meter. The mechanism around the disc is specified in the module "optic_disc". The module "detector" is responsible of the detection of a quantum, which is done by interpreting the different values of the position of the disc. The module "calendar" defines the precise moment and communicates it to the module "tariff_program". This module is responsible of calculating which tariff has to be applied at this moment and to deliver the current tariff to the module "distributor". This last module, if a quantum is detected, applies the current tariff to this quantum and sends it to the module "totalizer". This one calculates the sum of the different

applied tariffs.

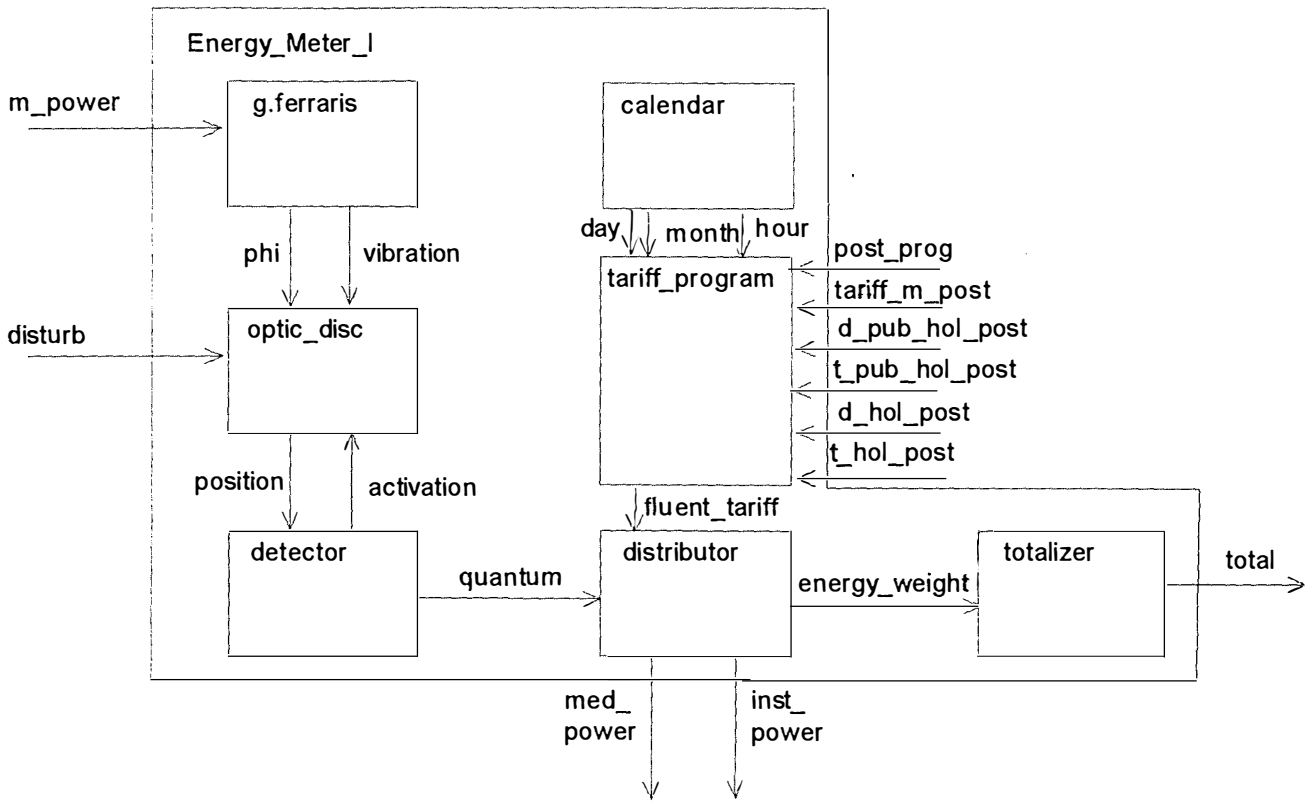

Figure 4.4: The graphic representation of the class Energy_Meter_I

The communication between the different modules is represented by the arrows on the graphic representations.

In the following specification, we focus on the modules "optic_disc" and "detector".

First, we look at the Energy_Meter with one photocell.

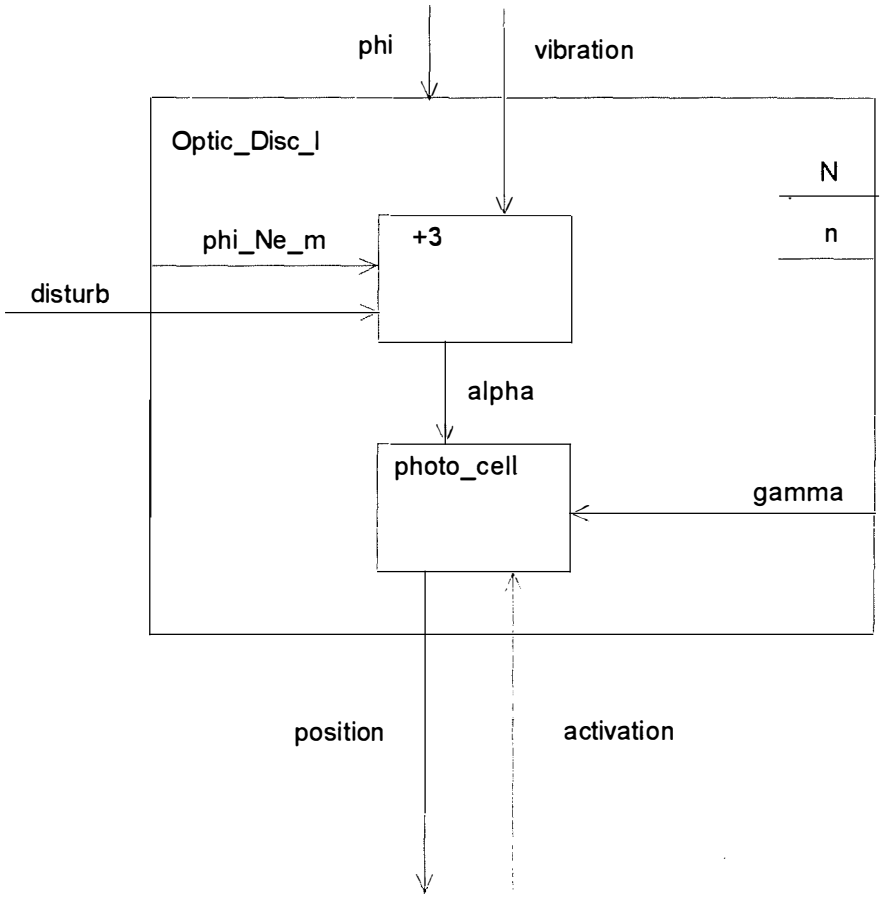

Figure 4.5: The graphic representation of the class Optic_Disc_I

The module "optic_disc_I" itself is composed of two other modules: "+3" and "photo_cell" as shown in figure 4.5. The first one defines the addition of 3 variables, in this case ϕ_Ne_m, disturb and vibration. The second one specifies the mechanism of the photocell, this module is represented in figure 4.6.

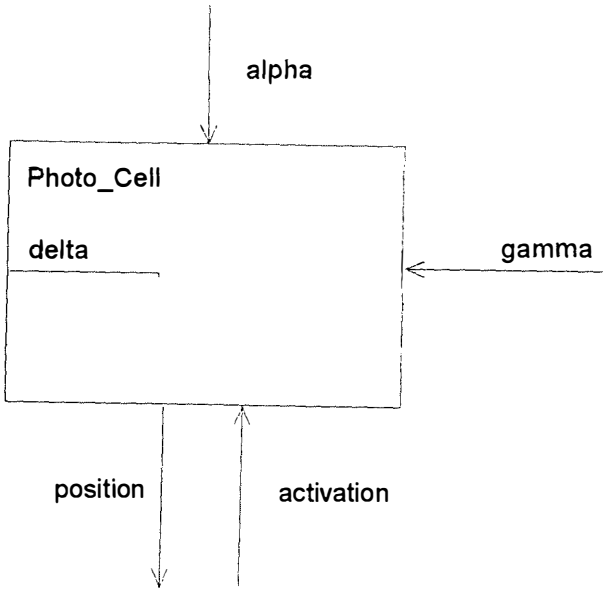

Figure 4.6: The graphic representation of the class Photo_Cell

The module "detector_I" has only one module "shift_register" as shown in figure 4.7. This module represents a register which is composed of a serie of little cells. If the first one receives a new value, the second cell gets the value of the first one and so on. Only the value of the last one is lost.

The main difference between the specification of the Energy_Meter with one photocell and the one with two photocells is stated in the module "optic_disc_II".

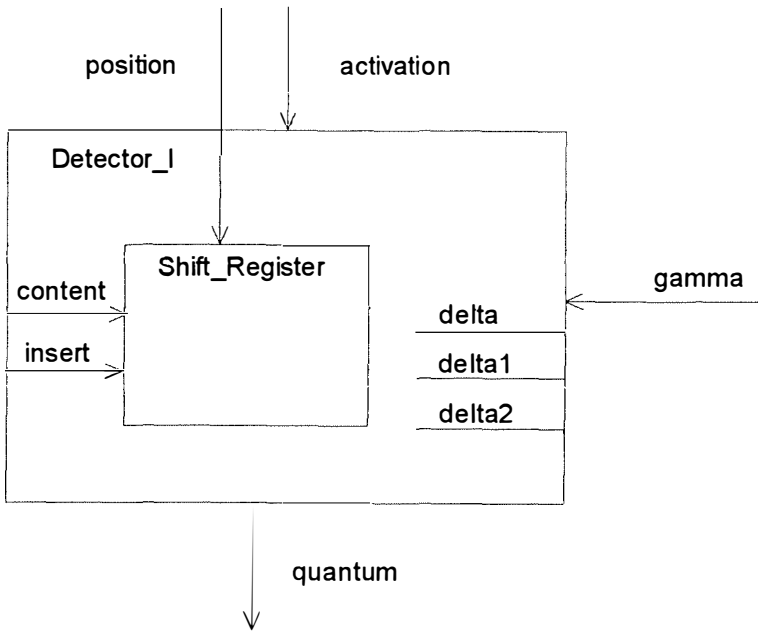

Figure 4.7: The graphic representation of the class Detector_I

But already at figure 4.8, the graphic representation of the class "Energy_Meter_II", we can see that the communication between "optic_disc_II" and "detector_II" has one more element than in the previous case: a 2nd position. We can also see that a new constant is introduced: shift. It represents the difference of the angle between the two photocells.

On figure 4.9, the graphic representation of "optic_disc_II" we notice that this module is now composed of four modules, two of them represent an addition of 2 and 3 variables, and each of the two other modules represents a photocell. One photocell works exactly the same way as in the previous case.

At the end of this section, 3 general classes are specified, which are inherited by different classes of the two specifications, the Energy_Meter with one photocell and the one with two photocells.

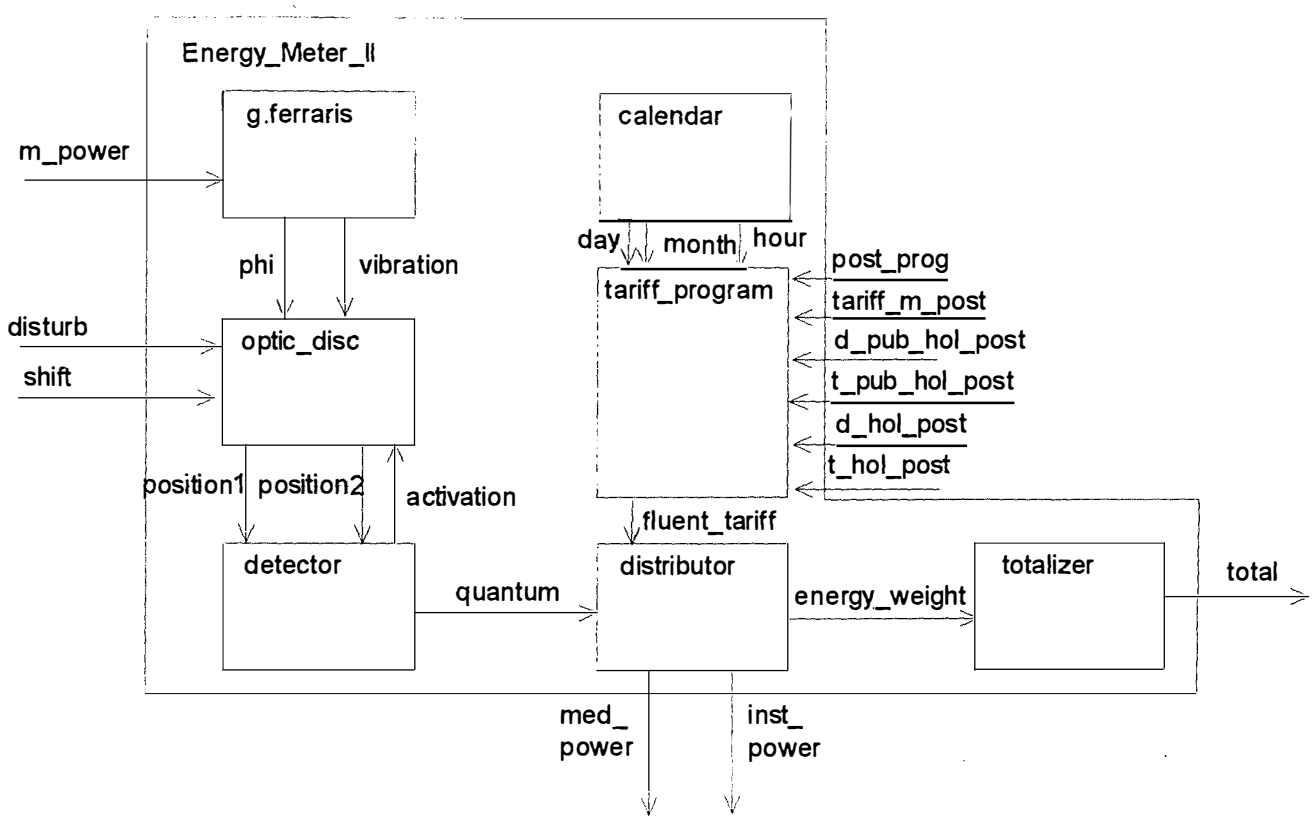

Figure 4.8: The graphic representation of the class Energy_Meter_II

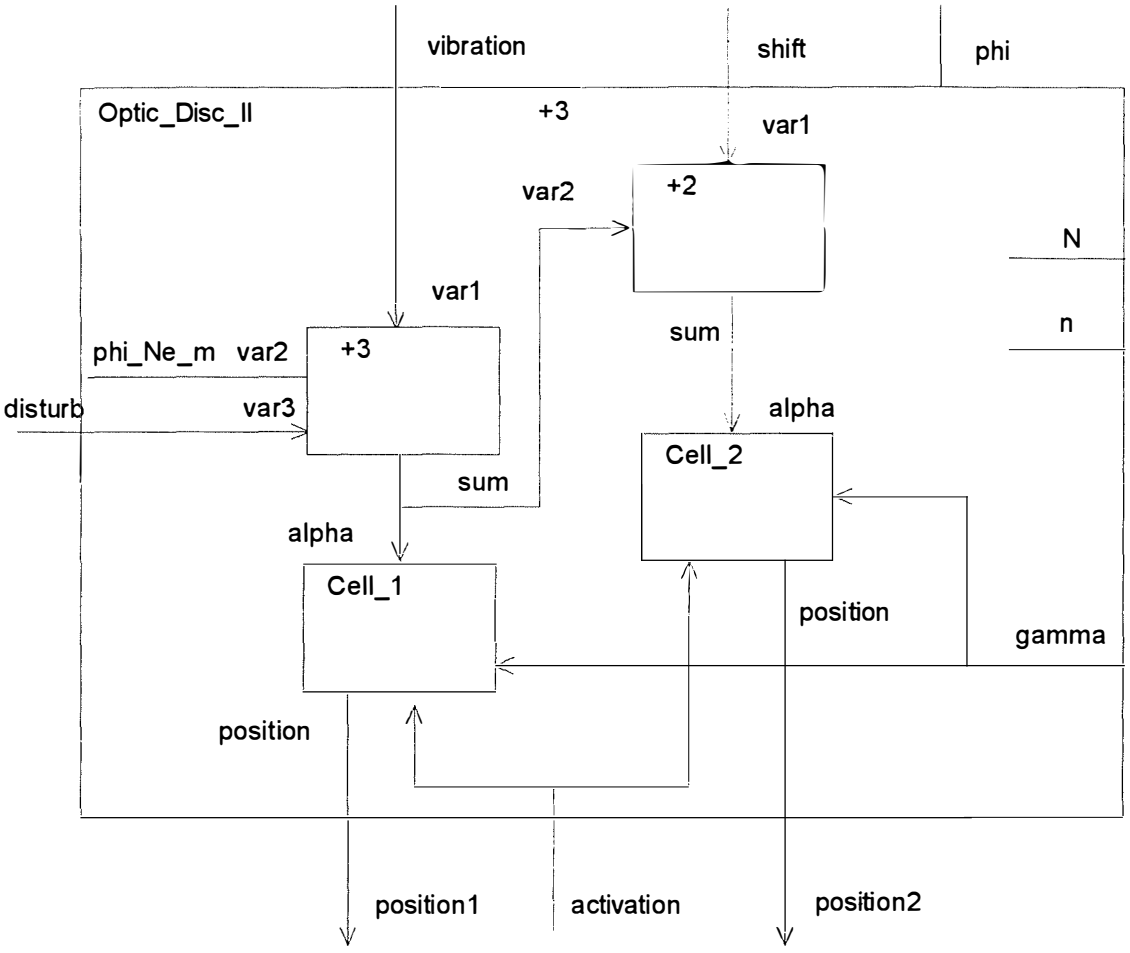

Figure 4.9: The graphic representation of the class Optic_Disc_II

Class SUM2TDVAR[VARTYPE]
general class of the addition of 2 variables of "varType"
Visible var1, var2, sum
TD Items
vars var1, var2, sum: varType
Axioms
sum = var1 + var2

end SUM2TDVAR

Class SUM3TDVAR[VARTYPE]
general class of the addition of 3 variables of "varType"
Visible var1, var2, var3, sum
TD Items
vars var1, var2, var3, sum: varType
Axioms
sum = var1 + var2 + var3

end SUM3TDVAR

Class DETECTOR
Visible position, activation, quantum
TI Items
consts δ_1, δ_2: real
TD Items
predicates position({F, V})
activation
quantum
confirm_transition
Axioms
Cycle:
Becomes(activation) $\rightarrow$
Lasts(activation, δ_1)
$\wedge$ Futr(NextTime(activation, δ_2), δ_1)
defines the cycle of activations

SendQuantum:
confirm_transition $\leftrightarrow$ quantum
a quantum is registered, when the system has detected a confirmed transition

end DETECTOR

4.2.1 The Energy_Meter with one photocell

The following specification is focused only on two of the modules of the Energy_Meter, the "Optic_Disc_I" and the "Detector_I". As shown on figure 4.5, the module "Optic_Disc_I" is composed of two other modules. These two modules, "Sum3TdReal" and "Photo_Cell" are specified, before we propose the specification of the "Optic_Disc_I". Further in this section, we give the specification of the "Shift_register" which is a module of the "Detector_I" as shown in figure 4.7. This specification is followed by the one of the "Detector_I".

Class SUM3TDREAL
is Sum3TdVar[Real]
instantiation of Sum3TdVar

end SUM3TDREAL

Class PHOTO_CELL
Visible α, γ, activation, position
TI Items
consts δ, γ: real
TD Items
predicates position({F, V})
activation
vars α: real
indicates the disc position
Axioms
r: F, V

Iff_activation:
$\exists$r position(r) $\leftrightarrow$ activation
the position of the disc is only evaluated, when the photocell is activated

Poss_lecture:

$$\frac{\delta}{2} \leq \frac{\gamma}{3}$$

this sufficient condition is guaranteeing that the Zone of Indecision is not too big

Lecture:

activation $\rightarrow$

$$\left\{ \begin{array}{c} \left\{ \begin{array}{c} 0 \leq \alpha \mathrm{mod} 2\gamma \leq \frac{\delta}{2} \\ \vee \\ \gamma - \frac{\delta}{2} \leq \alpha \mathrm{mod} 2\gamma \leq \gamma + \frac{\delta}{2} \\ \vee \\ 2\gamma - \frac{\delta}{2} \leq \alpha \mathrm{mod} 2\gamma \leq 2\gamma \end{array} \right\} \\ \rightarrow \{\mathrm{position(F)} \vee \mathrm{position(V)}\} \\ \wedge \\ (\frac{\delta}{2} < \alpha \mathrm{mod} 2\gamma < \gamma - \frac{\delta}{2}) \\ \rightarrow \mathrm{position(F)} \\ \wedge \\ (\gamma + \frac{\delta}{2} < \alpha \mathrm{mod} 2\gamma < 2\gamma - \frac{\delta}{2}) \\ \rightarrow \mathrm{position(V)} \end{array} \right\}$$

position(F) = opaque sector of the disc
position(V) = transparent sector of the disc
the value of the predicate "position" is stated in relation with the angle of the disc

end PHOTO_CELL

Class OPTIC_DISC_I
Visible disturb, position, activation, ϕ, vibration
TI Items
consts γ: real
n, N: integer
TD Items
predicates position({F, V})
activation
vars disturb: real
external cause of the system
vibration: real
intrinsic cause of the system

ϕ: real
position of the disc
ϕ_Ne_m: real
coeff. of the used energy_meter

Modules photo_cell: PHOTO_CELL
+3: SUM3RDREAL

Connections

{ disturb	+3.disturb
vibration	+3.vibration
ϕ_Ne_m	+3.ϕ_Ne_m
+3.α	photo_cell.α
γ	photo_cell.γ
activation	photo_cell.activation
photo_cell.position	position }

Axioms

PosDisc_ind:

$$\phi\text{\_Ne\_m} = \frac{\Phi}{N}$$

makes the position of the optic disc independent of the used energy_meter model

AngleOpaqueSector:

$$\gamma = \frac{2\Pi}{2n}$$

size of one opaque or transparent sector

NumOpaqueSector:

$$n = 10$$

the number of opaque sectors of the optic disc, the total number of sectors (opaque and transparent) is $2n$

end OPTIC_DISC_I

Class SHIFT_REGISTER
Visible position, content, insert
TI Items

vars content: all_1, all_0, mix
predicates position({F, V})
insert
slot([1..8])

Axioms

vars i: [1..8]
j: [1..7]

Initial_value:

$$\text{AlwP}(\neg\ \text{insert}) \rightarrow \forall \text{i}\ \neg\ \text{slot(i)}$$

the initial value of all the elements of the "slot" array is false

Postponement:

$$\text{insert} \rightarrow \left(\begin{array}{c} \text{position(F)} \rightarrow \text{Until\_ei(slot[8], insert)} \\ \wedge \text{position(V)} \rightarrow \text{Until\_ei}(\neg\text{slot[8]}, \text{insert}) \\ \wedge \forall \text{j} \left(\begin{array}{c} \text{slot[j + 1]} \rightarrow \text{Until\_ei(slot[j], insert)} \\ \wedge \neg\text{slot[j + 1]} \rightarrow \text{Until\_ei}(\neg\text{slot[j]}, \text{insert}) \end{array} \right) \end{array} \right)$$

if the predicate "insert" is true, then the update of the elements value of the "slot" array will be made and the new values will be memorized

Consistence:

$$(\forall\ \text{i}\ (\neg\ \text{insert} \wedge \text{slot[i]}) \rightarrow \text{slot[i]})$$

condition of consistence from one "insert" to the next one

Content_0:

$$(\forall\ \text{i}\ \neg\ \text{slot(i)}) \leftrightarrow (\text{content} = \text{all\_0})$$

if no element of the "slot" array is true, then "content" gets the value "all_0"

Content_1:

$$(\forall\ \text{i}\ \text{slot(i)}) \leftrightarrow (\text{content} = \text{all\_1})$$

if all the elements of the "slot" array are true, then "content" gets the value "all_1"

end SHIFT_REGISTER

Class DETECTOR_I
inherits DETECTOR
rename CONFIRM_TRANSITION **as** INSERT
redefine SENDQUANTUM

TI Items

consts δ: real

TD Items

vars content: all_1, all_2, mix
precState: 0, 1

Modules shift_register: SHIFT_REGISTER

Connections

{ content — shift_register.content
insert — shift_register.insert
position — shift_register.position }

Axioms

Const_0:

$\delta = 20\ \mu$sec

defines the instant of sampling

Const_1:

$\delta_1 = 25\ \mu$sec

defines the duration of the activation

Const_2:

$\delta_2 = 3{,}1$ msec

defines the interval between two activations

Insertion:

Becomes(activation) $\leftrightarrow$ Futr(insert,δ)

the sampling is registered $\delta = 20\mu$sec after the activation has started

Transition1:

Becomes(content = all_1)
$\rightarrow$ Until$_w$_ei(precState = 1, content = all_0)

when "content" becomes "all_1", means that the previous state will have the value 1 *until the value of "content" changes to "all_0"*

Transition2:

Becomes(content = all_0)
$\rightarrow$ Until$_w$_ei(precState = 0, content = all_1)

when "content" becomes "all_0", means that the previous state will

have the value 0 *until the value of "content" changes to "all_1"*

SendQuantum:
(Becomes(precState = 1) ∨ Becomes(precState = 0))
↔ quantum
a quantum is registered when a transition from a state "all_1" to a state "all_0" or vice-versa is detected

end DETECTOR_I

4.2.2 The Energy_Meter with two photocells

Like the specification of the Energy_Meter with one photocell, this specification is also focused on the "Optic_Disc_II" and the "Detector_II". As shown on figure 4.9, the module "Optic_Disc_II" is composed of three other modules. These three modules, "Sum2TdReal", "Sum3TdReal" and "Photo_Cell" are specified, before we specify the "Optic_Disc_II". Further in this section, we give the specification of the "Detector_II".

Class SUM2TDREAL
is Sum2TdVar[Real]
instantiation of Sum2TdVar
end SUM2TDREAL

Class SUM3TDREAL
is Sum3TdVar[Real]
instantiation of Sum3TdVar
end SUM3TDREAL

Class PHOTO_CELL
Visible α, γ, activation, position
TI Items
consts δ, γ: real
TD Items
predicates position({F, V})
activation
vars α: real
indicates the disc position

Axioms

r: F, V

Iff_activation:

$$\exists r \ \text{position}(r) \leftrightarrow \text{activation}$$

the position of the disc is only evaluated, when the photocell is activated

Poss_lecture:

$$\frac{\delta}{2} \leq \frac{\gamma}{3}$$

this sufficient condition is guaranteeing that the Zone of Indecision is not too big

Lecture:

$$\text{activation} \rightarrow \left\{ \begin{array}{l} \left\{ \begin{array}{c} 0 \leq \alpha \text{mod} 2\gamma \leq \frac{\delta}{2} \\ \vee \\ \gamma - \frac{\delta}{2} \leq \alpha \text{mod} 2\gamma \leq \gamma + \frac{\delta}{2} \\ \vee \\ 2\gamma - \frac{\delta}{2} \leq \alpha \text{mod} 2\gamma \leq 2\gamma \end{array} \right\} \\ \rightarrow \{\text{position(F)} \vee \text{position(V)}\} \\ \wedge \\ (\frac{\delta}{2} < \alpha \text{mod} 2\gamma < \gamma - \frac{\delta}{2}) \\ \rightarrow \text{position(F)} \\ \wedge \\ (\gamma + \frac{\delta}{2} < \alpha \text{mod} 2\gamma < 2\gamma - \frac{\delta}{2}) \\ \rightarrow \text{position(V)} \end{array} \right\},$$

position(F) = opaque sector of the disc
position(V) = transparent sector of the disc
the value of the predicate "position" is stated in relation with the angle of the disc

end PHOTO_CELL

Class OPTIC_DISC_II
inherits OPTIC_DISC_I
rename position **as** position1, photo_cell **as** cell_1
Visible shift, position2

TI Items

consts shift: real

shift represents the distance between the two photocells

TD Items

predicates position2({F, V})

Modules cell_2: PHOTO_CELL

+2: SUM3RDREAL

Connections

{ +3.sum	+2.var2
+2.sum	cell_2.α
shift	+2.var1
γ	cell_2.γ
cell_2.position	position2
activation	cell_2.activation }

Axioms

Angelshift:

$$\text{shift} = \frac{\Phi}{2n}$$

$$\textit{shift} = \frac{\gamma}{2} + n\gamma$$

end OPTIC_DISC_II

Class DETECTOR_II
inherits DETECTOR
rename POSITION **as** POSITION1, CONFIRM_TRANSITION **as** BUP1
redefine SENDQUANTUM
Visible position2
TD Items

predicates position({F, V})

bup1

begin of the rise of signal 1

bdw1

begin of the descend of signal 1

bup2

begin of the rise of signal 2

bdw2

begin of the descend of signal 2

bupconf1

begin of the confirmation of the rise of signal 1

bdwconf1
begin of the confirmation of the descend of signal 1

Axioms

BeginUp1:

bup1 $\leftrightarrow$ position1(F)
$\wedge$ Since($\neg$ position1(F), position1(V))

"bup1" is true when photocell1 is in front of an opaque sector and when photocell1 had not been in front of an opaque sector since it had been in front of a transparent one

BeginDw1:

bdw1 $\leftrightarrow$ position1(V)
$\wedge$ Since($\neg$ position1(V), position1(F))

"bdw1" is true when photocell1 is in front of a transparent sector and when photocell1 had not been in front of a transparent sector since it had been in front of an opaque one

BeginUp2:

bup2 $\leftrightarrow$ position2(F)
$\wedge$ Since($\neg$ position2(F), position2(V))

"bup2" is true when photocell2 is in front of an opaque sector and when photocell2 had not been in front of an opaque sector since it had been in front of a transparent one

BeginDw2:

bdw2 $\leftrightarrow$ position2(V)
$\wedge$ Since($\neg$ position2(V), position2(F))

"bdw2" is true when photocell2 is in front of a transparent sector and when photocell2 had not been in front of a transparent sector since it had been in front of an opaque one

BeginUpCf1:

bupconf1 $\leftrightarrow$ bup2 $\wedge$ Since($\neg$ bup2, bup1)

"bupconf1" is true when "bup2" is true and when "bup2" had not been true since "bup1" had been true

BeginDwCf1:

$$\text{bdwconf1} \leftrightarrow \text{bdw2} \wedge \text{Since}(\neg\ \text{bdw2},\ \text{bdw1})$$

"bdwconf1" is true when "bdw2" is true and when "bdw2" had not been true since "bdw1" had been true

SendQuantum:

$$\text{quantum} \leftrightarrow (\text{bupconf1} \vee \text{bdwconf1})$$

a quantum is only registered when a confirmed transition from an opaque to a transparent sector or vice-versa had been detected

end DETECTOR_II

Chapter 5

The ALBERT Specification of the Energy_Meter

In this chapter we propose a second specification of the 'Energy_Meter' case study. This specification is realized in ALBERT [5, 6] (an Agent-oriented Language for Building and Elicitating Requirements for Real Time Systems) developed at the University of Namur.

ALBERT is a "language for modelling the functional requirements", "based on a variant of temporal logic, a mathematical language particularly suited for describing traces [1] " [5], to which three extensions are added:

- the concept of "actions" is introduced to overcome the frame problem [2],
- the concept of "agents" which can be seen as a specialization of the concept of objects and as way of structuring a specification,
- the concept of "template", which should guide the analyst in specifying a system.

[1] This mathematical language is an "extension of the multi-sorted first-order logic."

[2] The frame problem is "to mention explicitly in the specification not just the things that are changed by the procedure, but also all those that are not (...stating that "nothing else changes") [2].

5.1 Specification in ALBERT

For the specification in ALBERT, we adopte another approach, but we focused on the same part of the specification, the "OpticDisc" and the "Detector". In the ALBERT specification, "OpticDisc" and "Detector" are two agents.

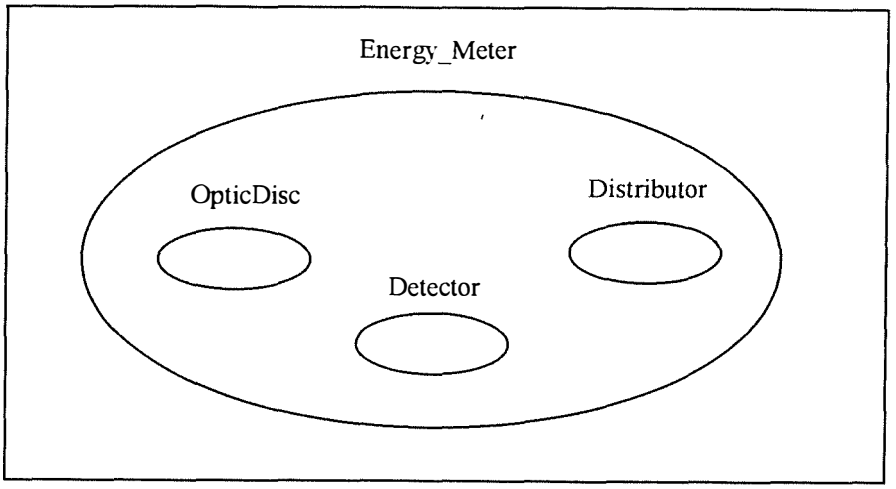

Figure 5.1: The graphic representation of the Society

The Society of the Energy_Meter, as shown in figure 5.1, is composed of three agents, the "OpticDisc", the "Detector" and the "Distributor". The last agent, the "Distributor", is not specified in this document. The Agent "Detector" is responsible for the activation of the light of the photocell and also to find out if there was energy consumption between the last 2 activations by analyzing the informations furnished by the Agent "OpticDisc". It has to inform the Agent "Distributor" of the consumed quantums, this agent is not modularized here. The Agent "OpticDisc" has to inform the Agent "Detector" if, during the activation of the photocell, it was able to 'see' the light signal or not. The Agent "DetectorI" and "OpticDiscI" are respectively the "Detector" and "OpticDisc" of the Energy_Meter with one photocell and are represented graphically in figure 5.2 and 5.3.

The case of the Energy_Meter with two photocells is similar, the only difference is that several of the information related to the photocells are doubled, one for photocell1 and the other for photocell2. The graphic representation of the two agents for the Energy_Meter with two photocells are shown in figure 5.4 and 5.5.

5.1.1 The Energy_Meter with one photocell

In this section, we specify first the agent "DetectorI" and then the agent "OpticDiscI" in ALBERT. But before starting with the specification, we have to define several operations:

Operation ZoneI: RATIONAL $\rightarrow$ BOOLEAN

ZoneI(r) = b with b $\Leftrightarrow$ $\begin{pmatrix} (0 \leq r \leq \frac{\delta}{2}) \\ \vee(\gamma - \frac{\delta}{2} \leq r \leq \gamma + \frac{\delta}{2} \\ \vee(2\gamma - \frac{\delta}{2} \leq r \leq 2\gamma) \end{pmatrix}$

Operation ZoneT: RATIONAL $\rightarrow$ BOOLEAN

ZoneT(r) = b with b $\Leftrightarrow$ $(\frac{\delta}{2} < r < \gamma - \frac{\delta}{2})$

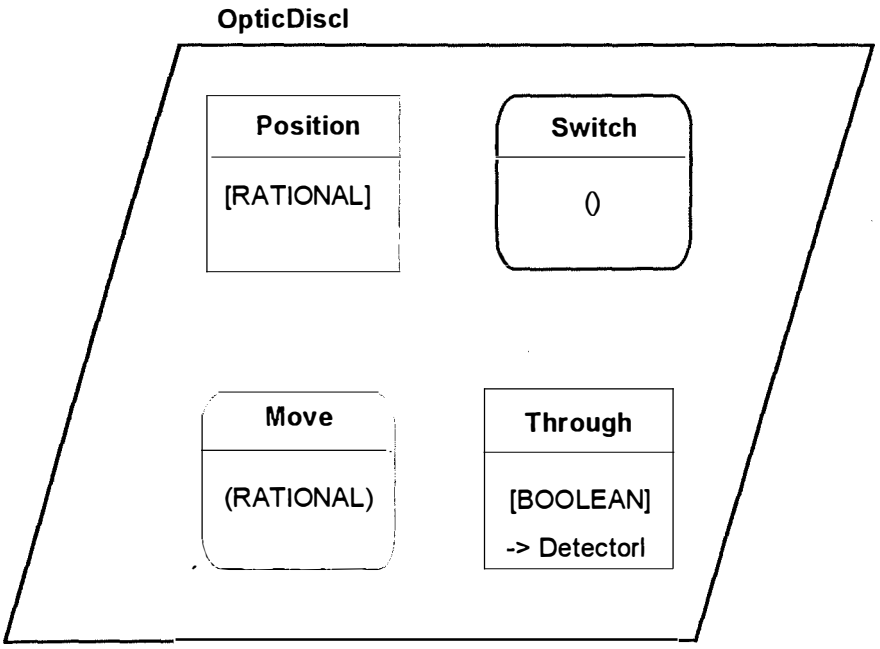

Figure 5.2: The graphic representation of OpticDiscI

AGENT OpticDiscI
DECLARATIONS
STATE COMPONENTS
Position instance_of RATIONAL
"Position" is the angle of the photocell to the disc
Through instance_of BOOLEAN → DetectorI
"Through" is true when the receiver of the photocell receives a light signal.
ACTIONS
Move(RATIONAL)
The action "Move" represents the vibration of the disc due to external causes.
*Switch
"Switch" is the action when the part of the disc in front of the photocell switches from an opaque sector to a transparent one or vice-versa.
DECLARATIVE CONSTRAINTS
STATE BEHAVIOUR
¬DetectorI.Alight ⇒ ¬Through
If the photocell is not activated then the receiver of the photocell can not get a light signal.
¬ZoneI(Position) ⇒ Through = (DetectorI.Alight ∧ ZoneT(Position))
If the photocell is not in front of a ZoneI of the disc then the photocell can only receive a light signal when the photocell is activated and when the part of the disc in front of the photocell is a transparent sector.
OPERATIONAL CONSTRAINTS
EFFECT OF ACTIONS
Switch: [] Through := ¬Through
Move(r): [] Position := (Position + r) mod (2γ)
COOPERATION CONSTRAINTS
STATE INFORMATION
K(Through.DetectorI/True)
"DetectorI" may always check "through".
STATE PERCEPTION
K(DetectorI.Alight/True)
"PhotoCellI can always look at "Alight".

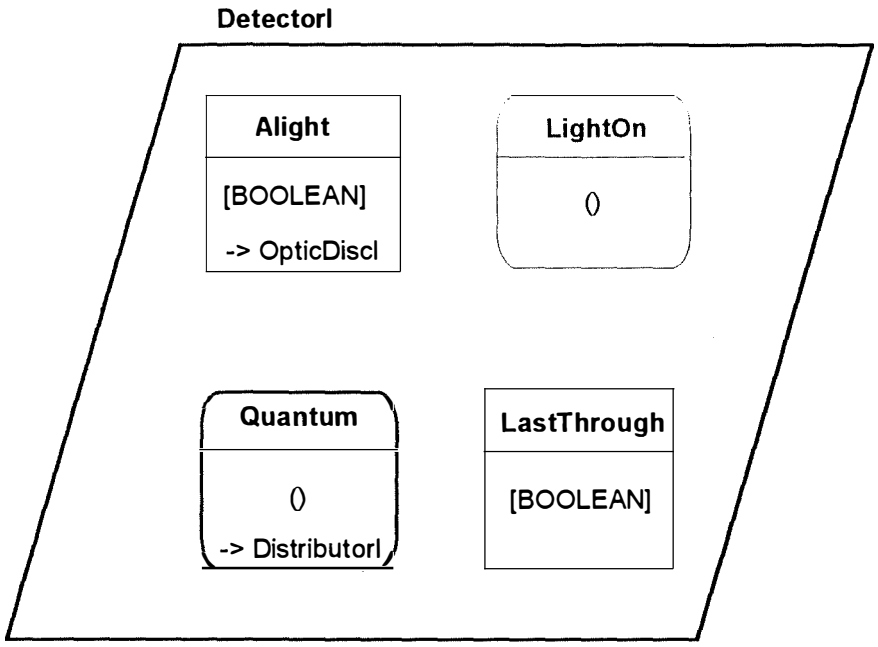

Figure 5.3: The graphic representation of DetectorI

AGENT DetectorI
DECLARATIONS
STATE COMPONENTS
Alight instance_of BOOLEAN → OpticDiscI
"Alight" is true when the photocell is activated.
LastThrough instance_of BOOLEAN
"LastThrough" is true if "through" was true when the last quantum was detected.
ACTIONS
LightOn
"LightOn" is the action of activating the photocell.
*Quantum → Distributor
"Quantum" is the detection of one unit of energy consumption.
BASIC CONSTRAINTS
INITIAL VALUATION
Alight = False
LastThrough = False

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

[LightOn] Alight Until(Lasts$_{\delta_2}$ $\neg$Alight $\wedge$ Futr$_{\delta_2}$ Alight)

Only during the action "LightOn", "Alight" is true and until "Alight" will be true again in δ_2 time units, "Alight" will be false during δ_2 time units.

ACTION DURATION

|LightOn| = δ_1

The action "LightOn" lasts δ_1 time units.

OPERATIONAL CONSTRAINTS

PRECONDITIONS

Quantum: Lasted$_{\delta_1}$ (Alight $\wedge$ (OpticDiscI.Through $\neq$ LastThrough)

The action "Quantum" can only take place when during δ_1 time units, the photocell is activated and when the part of the disc in front of the photocell is different from the part in front of the photocell when the last quantum was detected.

EFFECT OF ACTIONS

Quantum: [] LastThrough := OpticDiscI.Through

LightOn: Alight := True [] Alight := False

TRIGGERINGS

Alight $\wedge$ (OpticDiscI.Through $\neq$ LastThrough) / δ_1 $\rightsquigarrow$ Quantum

The action "Quantum" takes place when the photocell is activated and when the part in front of the photocell is different from the part in front of the photocell when the last quantum was detectec, during δ_1 time units.

COOPERATION CONSTRAINTS

STATE INFORMATION

K(Alight.OpticDiscI/True)

The "OpticDiscI" may always check "Alight".

STATE PERCEPTION

K(OpticDiscI.Through/True)

The "DetectorI" can always look at "Through".

ACTION INFORMATION

K(Quantum.Distributor/True)

The "DetectorI" always sends a message to the "Distributor" when a quantum is detected.

We only show here, that it is possible to see a problem intuitively in the ALBERT specification. The "Move" action is due to the vibration of the mechanism and the effect of this action is that the position will be modified. As there are no constraints on the way of the modification, it can modify the position backwards and forwards. If the position of the disc in front of the photocell is near the frontier of a sector. the modification backwards and forwards of the position can be such that the part in front of the photocell is alternatively a transparent sector and an opaque one. The fact that the sector in front of the photocell changes is represented by the action "Switch" and it gives the impression of energy consumption to the "DetectorI" agent.

So, the vibration of the mechanism can cause the detection of one or several quantums. The Energy_Meter with one photocell is not very rigorous, as it is proved for the TRIO+ specification in the following chapter.

5.1.2 The Energy_Meter with two photocells

As for the Energy_Meter with one photocell, we start the specification of the Energy_Meter with two photocell with the definition of several operations. Further on, we propose the ALBERT specification of the agent "DetectorII" and the agent "OpticDiscII".

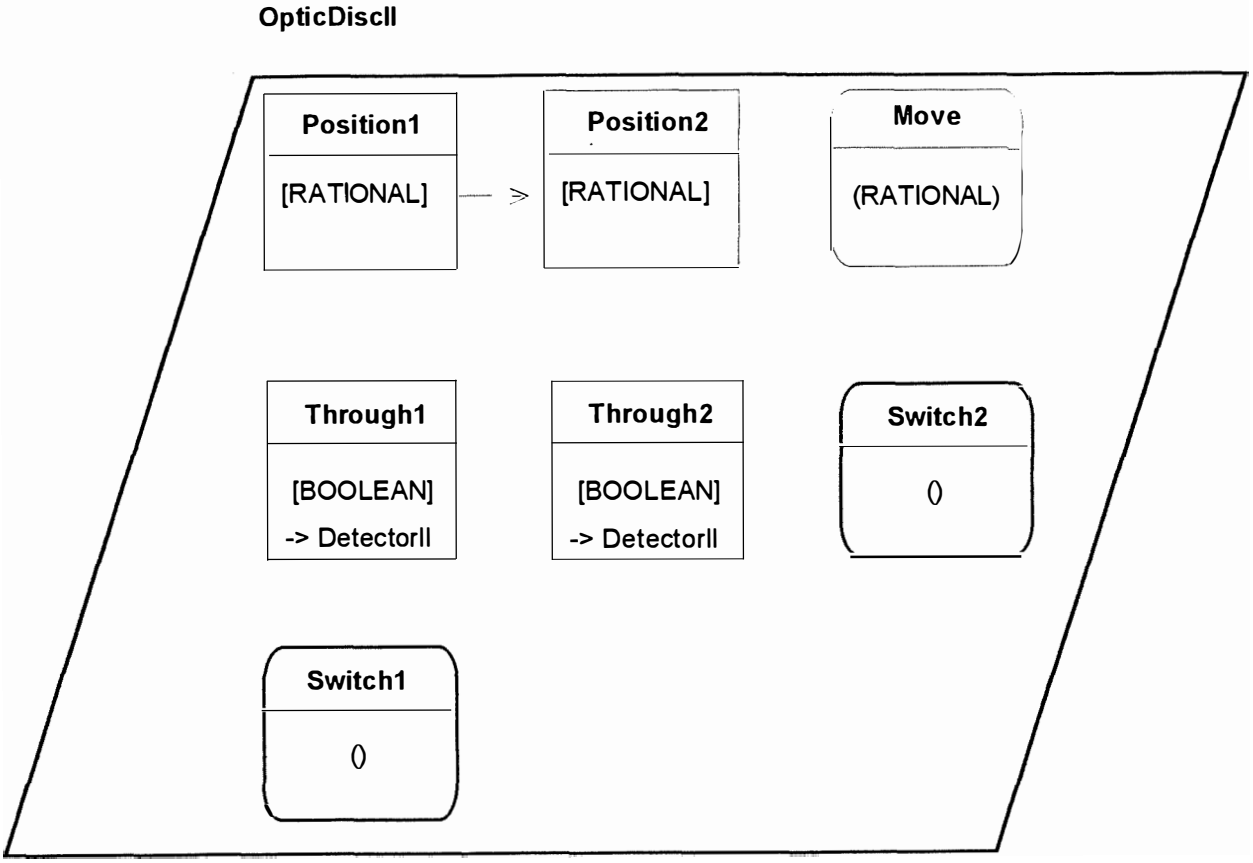

Figure 5.4: The graphic representation of OpticDiscII

Operation ZoneI: RATIONAL → BOOLEAN

ZoneI(r) = b with b $\Leftrightarrow \left(\begin{array}{c} (0 \leq r \leq \frac{\delta}{2}) \\ \vee(\gamma - \frac{\delta}{2} \leq r \leq \gamma + \frac{\delta}{2} \\ \vee(2\gamma - \frac{\delta}{2} \leq r \leq 2\gamma) \end{array} \right)$

Operation ZoneT: RATIONAL → BOOLEAN

ZoneT(r) = b with b $\Leftrightarrow (\frac{\delta}{2} < r < \gamma - \frac{\delta}{2})$

AGENT OpticDiscII

DECLARATIONS

STATE COMPONENTS

Position1 instance_of RATIONAL

"Position1" is the angle of photocell1 of the disc.

Position2 instance_of RATIONAL

"Position2" is the angle of photocell2 to the disc.

Through1 instance_of BOOLEAN → DetectorII

"Through1" is true when the receiver of photocell1 receives a light signal.

Through2 instance_of BOOLEAN → DetectorII

"Through2" is true when the receiver of photocell2 receives a light signal.

ACTIONS

Move(RATIONAL)

The action "Move" represents the vibration of the disc due to external causes.

*Switch1

"Switch1" is the action when the part of the disc in front of photocell1 switches from an opaque sector to a transparent one or vice-versa.

*Switch2

"Switch2" is the action when the part of the disc in front of photocell2 switches from an opaque sector to a transparent one or vice-versa.

BASIC CONSTRAINTS

DERIVED COMPONENTS

Position1 = Position2 + nγ + $\frac{\gamma}{2}$

The difference between the position of photocell1 and the position of photocell2 is the distance of n sectors and a half.

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

¬DetectorII.Alight ⇒ ¬Through1

If photocell1 is not activated then the receiver of photocell1 can not get

a light signal.
¬ZoneI(Position1) ⇒ Through1 = (DetectorII.Alight ∧ ZoneT(Position1))
If photocell1 is not in front of a ZoneI of the disc then photocell1 can only receive a light signal when photocell1 is activated and when the part of the disc in front of photocell1 is a transparent sector.
¬DetectorII.Alight ⇒ ¬Through2
If photocell2 is not activated then the receiver of photocell2 can not get a light signal.
¬ZoneI(Position2) ⇒ Through2 = (DetectorII.Alight ∧ ZoneT(Position2))
If photocell2 is not in front of a ZoneI of the disc then photocell2 can only receive a light signal when photocell2 is activated and when the part of the disc in front of photocell2 is a transparent sector.

OPERATIONAL CONSTRAINTS

EFFECT OF ACTIONS

Switch1: [] Through1 := ¬Through1
Switch2: [] Through2 := ¬Through2
Move(r): [] Position1 := (Position1 + r) mod (2γ)

COOPERATION CONSTRAINTS

STATE INFORMATION

K(Through1.DetectorII/True)
"DetectorII" may always check "through1".
K(Through2.DetectorII/True)
"DetectorII" may always check "through2".

STATE PERCEPTION

K(DetectorII.Alight/True)
"PhotoCellII" can always look at "Alight".

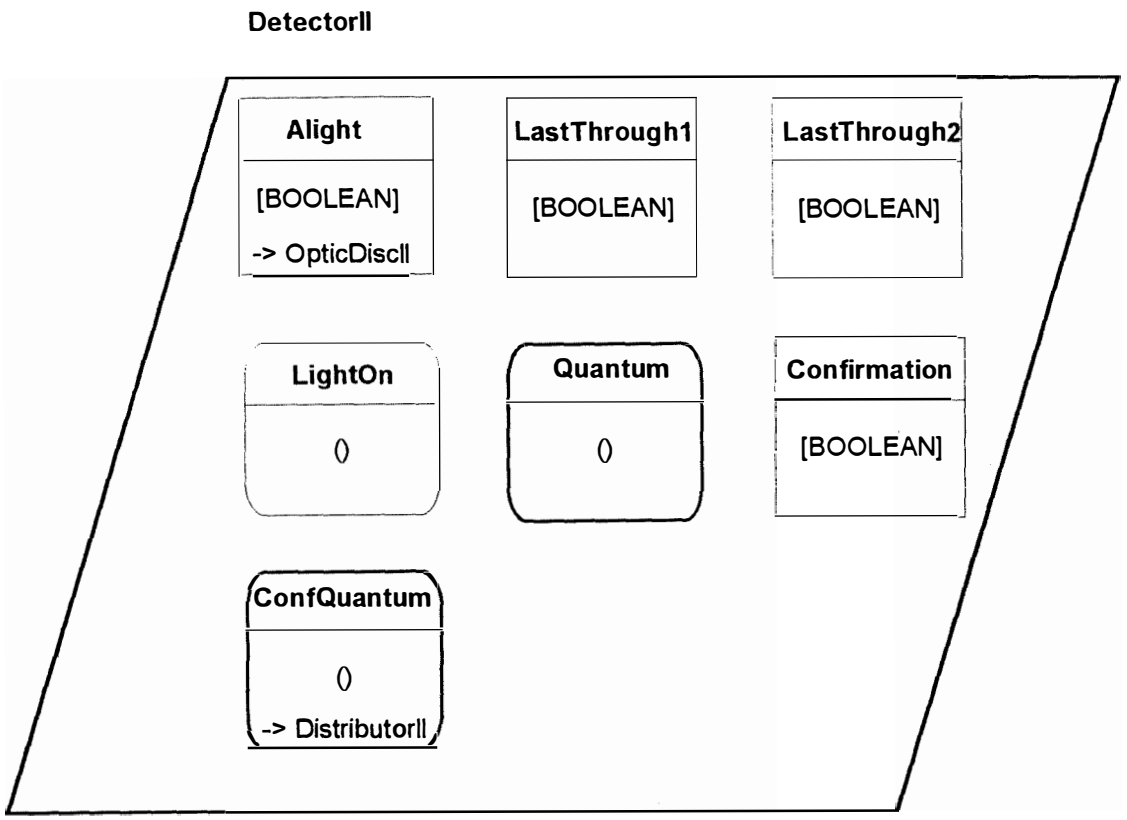

Figure 5.5: The graphic representation of DetectorII

AGENT DetectorII
DECLARATIONS
STATE COMPONENTS
Alight instance_of BOOLEAN → OpticDiscII
"Alight is true when the photocells are activated"
LastThrough1 instance_of BOOLEAN
"LastThrough1" is true if "through1" was true when the last quantum was detected.
LastThrough2 instance_of BOOLEAN
"LastThrough2" is true if "through2" was true when the last quantum was confirmed.

Confirmation instance_of BOOLEAN
"Confirmation is true when photocell1 has detected a quantum which has to be confirmed.

ACTIONS

LightOn
"LightOn" is the action af activating the photocells.
*Quantum
"Quantum" is the action that photocell1 has detected a quantum, which has to be confirmed.
*ConfQuantum → Distributor
"ConfQuantum" is the action that photocell2 has confirmed a quantum detected by photocell1.

BASIC CONSTRAINTS

INITIAL VALUATION

Alight = False
LastThrough1 = False
LastThrough2 = False
Confirmation = False

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

[LightOn] Alight Until(Lasts$_{\delta_2}$ ¬Alight ∧ Futr$_{\delta_2}$ Alight)
Only during the action "LightOn", "Alight" is true and until "Alight" will be true again in δ_2 time units, "Alight" will be false during δ_2 time units.

ACTION DURATION

|LightOn| = δ_1
The action "LightOn" lasts δ_1 time units.

OPERATIONAL CONSTRAINTS

PRECONDITIONS

Quantum: Lasted$_{\delta_1}$ (Alight ∧ (OpticDiscII.Through1 ≠ LastThrough1)
The action "Quantum" can only take place when during δ_1 time units, photocell1 is activated and when the part of the disc in front of photocell1 is different from the part in front of photocell1 when the last quantum was detected.

ConfQuantum: Confirmation
∧ Lasted$_{\delta_1}$ (Alight ∧ (OpticDiscII.Through2 ≠ LastThrough2)
The action "ConfQuantum" can only take place when photocell1 has detected a quantum which has still to be confirmed and when during δ_1

time units the photocell2 is activated and the part of the disc in front of photocell2 is different from the part in front of photocell2 when the last quantum was confirmed.

EFFECT OF ACTIONS

Quantum: [] LastThrough1 := OpticDiscII.Through1
Confirmation := True

ConfQuantum: [] LastThrough2 := OpticDiscII.Through2
Confirmation := False

LightOn: Alight := True
[] Alight := False

TRIGGERINGS

Alight $\wedge$ (OpticDiscII.Through1 $\neq$ LastThrough1) / $\delta_1 \rightsquigarrow$ Quantum
The action "Quantum" takes place when photocell1 is activated and when the part in front of photocell1 is different form the part in front of photocell1 when the last quantum, which has to be confirmed, was detected, during δ_1 time units.

Confirmation $\wedge$ Alight
$\wedge$ (OpticDiscII.Through2 $\neq$ LastThrough2) / $\delta_1 \rightsquigarrow$ ConfQuantum
The action "ConfQuantum" takes place when photocell1 has detected a quantum which has still to be confirmed and when photocell2 is activated and the part of the disc in fornt of photocell2 is different from the part in front of photocell2 when the last quantum was confirmed, during δ_1 time units.

COOPERATION CONSTRAINTS

STATE INFORMATION

K(Alight.OpticDiscII/True)
The "OpticDiscII" may always check "Alight".

STATE PERCEPTION

K(OpticDiscII.Through1/True)
The "DetectorII" can always look at "Through1".
K(OpticDiscII.Through2/True)
The "DetectorII" can always look at "Through2".

ACTION INFORMATION

K(ConfQuantum.Distributor/True)
The "DetectorII" always sends a message to the "Distributor" when a quantum is detected and confirmed.

5.2 An evaluation of the two specifications

The two specification languages, ALBERT and TRIO+, are based on real-time logic. We first analyze the particularities of the two languages and then we compare them in a more global view.

- A first particularity is the structure of an ALBERT specification. In order to guide the analyst in structuring the properties of a system, 12 different templates regrouped in 4 families are proposed. The four families are:

  1. Basic Constraints
  2. Declarative Constraints
  3. Operational Constraints
  4. Cooperation Constraints

  These templates are able to cover all the possible properties of a given system. The structure of an ALBERT specification is never composed of another template then those which are predefined in the language, but not all of them must be used.

  TRIO+ proposes no predefined template. It is up to the analyst to choose the most adapted style for his specification. Nevertheless, in the declaration part, some guidance is proposed. The different variables and functions are devided into Time Dependent and Time Independent ones.

- Another particularity of ALBERT is the possibility to express dynamic properties by the concept of actions. An action can be [6]:

  1. an happening having an effect on the state
  2. an happening with no direct influence on the state

  A sequence of different actions and states define a possible life of the agent.

  Compared to this, TRIO+ is a specification language in which a system is described in a static way.

  An important advantage of the concept of action is that it solves the frame problem. In the ALBERT specification only the effects of an

action are described and the "nothing else changes" is implicit. As all the frame axioms have not to be established, the specification stays shorter and more perspicuous [2].

- As we have mentioned in the first chapter, TRIO+ is an object-oriented language. The OO concepts offer the possibility of structuring a specification of a complex system, e.g. 'a specification in the large'.

  In ALBERT, the concept of agents is introduced. This concept can be seen as a specialization of the OO concept of an object and as such is used to structure the specification. The difference between these two concepts is that contractual responsibilities are attached to each agent. In an interaction between agents, each agent is not only responsible of which state and/or action it informs its interlocutor, but also which of the information from the other agents it perceives.

- TRIO+ offers the possibility of inheritance. But it is left up to the analyst to make sure that the inheritance is not only syntactic but also semantic.

  In ALBERT, another choice is adopted. As it is difficult to guarantee semantic inheritance, no inheritance at all is allowed in the actual version, in order to avoid purely syntactic inheritance.

- TRIO+ supports a genericity mechanism. As in the previous case, it is up to the analyst to ensure the semantic use of this mechanism.

  In ALBERT, the same choice as for the inheritance mechanism has been adopted, no genericity mechanism is allowed until it can be guaranteed that the genericity is not only syntactic.

After having seen all these characteristics, we can conclude, that ALBERT is a strongly structured specification language, which intends to guide the analyst methodically to a consistent and complete specification.

In opposition to this, TRIO+ leaves more possibilities to the analyst, because it is up to him to choose the structure of the specification, which seems to be the best for each case. There is a risk, that there is no structure at all, if the analyst is not rigorous enough.

Even if ALBERT and TRIO+ are specification languages we can say that a ALBERT specification is much more abstract than a TRIO+ one. The ALBERT language offers the possibility to the analyst to specify a system

without referring to a special design model or implementation. This has the advantage that an ALBERT specification is not dependent on the further development of the system, e.g. even if all the design has to be changed, the ALBERT specification stays the same. In TRIO+, we can already find some ideas of a later implementation of the system. This leads us to say, that the two specification languages are not on the same level of abstraction. The ALBERT specification is better for formalizing a requirements document and later on a TRIO+ specification can be interesting, because the developers can introduce some implementation options without referring to a specific programming language.

Chapter 6

Proving a TRIO+ specification in PVS

6.1 Introduction

In the previous chapter we have proposed a specification of an Energy_Meter with one photocell and one with two photocells. In this chapter, we demonstrate that the Energy_Meter with one photocell is not rigorous enough in its way to detect a quantum. The falsification consists in the fact to show that the quantums counted by this Energy_Meter are not necessarily due to energy consumption.

Further more, we prove that the Energy_Meter with two photocells is reliable. The verification is devided in two parts. We demonstrate that the mechanism with two photocells counts **only** quantums due to energy consumption and that it counts **all** of them. To prove this last statement, we show that between two consecutive detected quantums, the rotation of the disc is an angle correspondent to one part of the disc, a transparent or an opaque one, $(\gamma \pm \delta)$.

At the end of this chapter, we show the system's behaviour in some situations it can be faced to, we have called them properties. The proof of one of these properties are explained in detail.

6.2 Falsification of the Energy_Meter with one photocell

6.2.1 The detection of a quantum

In the case of the Energy_Meter with one photocell, a quantum is counted when the receiver registers a movement of the disc. As the disc has in its outerpart evenly distributed opaque and transparent sections, a transition from an opaque to a transparent section or from a transparent to an opaque one indicates a movement of the disc. Unfortunately, with this method of detecting energy consumption, the receiver can register quantums without a movement of the disc due to energy consumption.

To illustrate one of these possibilities, we consider the case where, in the moment of no consumption of energy, the part of the disc in front of the photocell is just on the frontier part of transparent and opaque sector. The vibration of the Energy_Meter itself can cause a slightly movement back and forth of the disc and this movement may give the impression of energy consumption to the Energy_Meter. A quantum, whose detection is not due to a disc movement, not caused by energy consumption but to the vibration, is counted.

6.2.2 The falsification

To prove that a theorem is not valid, it is sufficient to find one particular situation in which the theorem is not valid.

In this proof, we intend to show that only the fact of a certain position of the disc is sufficient to have a quantum. This position is defined by the mechanism of the Energy_Meter itself. As mentionned before, in the regions, $0 \leq \alpha \bmod 2\gamma \leq \frac{\delta}{2}$, $\gamma - \frac{\delta}{2} \leq \alpha \bmod 2\gamma \leq \gamma + \frac{\delta}{2}$ and $2\gamma - \frac{\delta}{2} \leq \alpha \bmod 2\gamma \leq 2\gamma$, designed as Zone of Indecision, we are not sure which result we obtain, it is either position(F) (opaque sector) **or** position(V) (transparent sector).

In this case it is possible, even if the disc does not move at all, to have once a position(F) and then a position(V) as result of an activation of the photocell. This transition represents a quantum to our system.

6.2.3 The proof in PVS

The following theorem states that the fact of having the part of the disc in front of the photocell which is a ZoneI, is sufficient to have a quantum.

```
Falsif: THEOREM
    Alw((ZoneI & TD_alpha(alpha) & activation
    => (content_0 OR content_1))
    & ((UpToNow(NOT content_0) & content_0) => quantum)
    & ((UpToNow(NOT content_1) & content_1) => quantum))
```

The figure 6.1 shows the different proof steps.

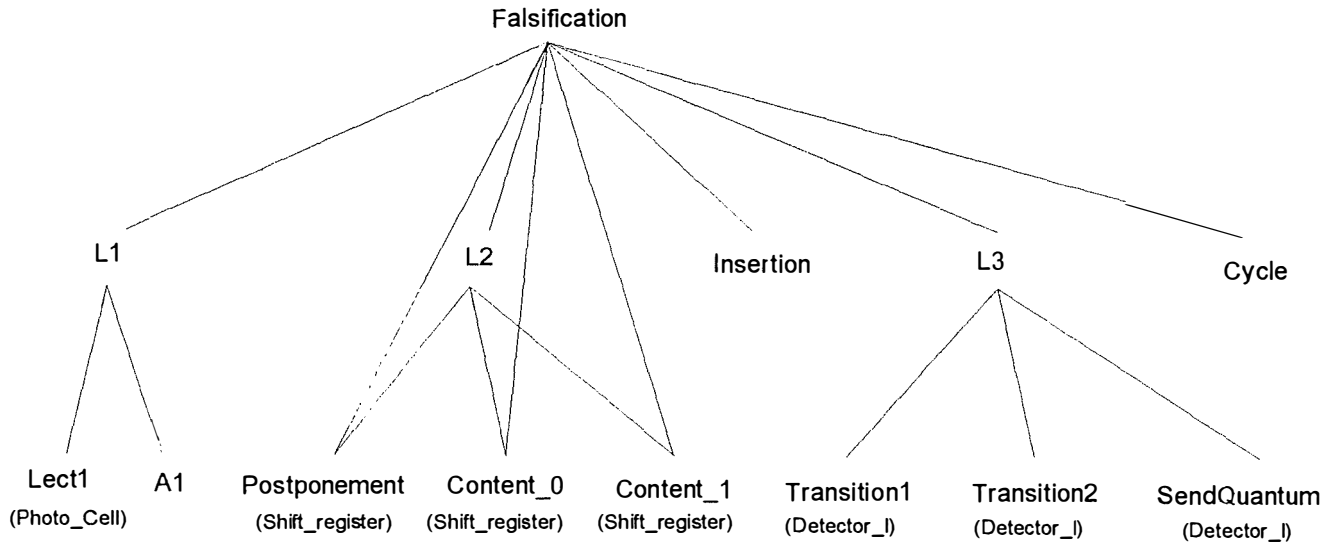

Figure 6.1: The proof-tree of the falsification

A Theorem or Lemma is represented as a node and they are provable by the lemmas and axioms which are represented as the 'sons' of the node. Following this diagram, the theorem "Falsif" is provable with "L1", "L2", "L3" and "Insertion" of the "Detector_I" Theory.

To prove the different lemmas in PVS, we need first of all to define the Zone of Indecision (ZoneI). This is done by the following axiom:

```
A1: AXIOM
    Alw(ZoneI & TD_alpha(alpha)
    <=> ((EXISTS n: 2 n gamma <= alpha
        & alpha <= 2 n gamma + delta / 2)
    OR(EXISTS n: 2 n gamma + gamma - delta / 2 <= alpha
        & alpha <= 2 n gamma + gamma + delta / 2)
    OR(EXISTS n: 2 n gamma + 2 gamma - delta/2 <= al
        & alpha <= 2 n gamma + 2 gamma)))
```

The following lemma is stating that the ZoneI gives as result position(V) or position(F).

```
L1: LEMMA
    Alw((ZoneI & TD_alpha(alpha) & activation)
    => position(V) OR position(F))
```

Now, to complete our proof, we need only to prove that if the position changes, the system detects a quantum. This is stated by L2 and L3:

```
L2: LEMMA
    Alw(((position(V) & insert) OR (position(F) & insert))
    => content_0 OR content_1)

L3: LEMMA
    Alw((UpToNow(NOT content_0) & content_0)
        OR (UpToNow(NOT content_1) & content_1)
    => quantum)
```

6.3 Verification and different properties of the Energy_Meter with two photocells

6.3.1 The detection of a quantum

To avoid counting of quantums which are not due to energy consumption, a similar system with two photocells has been designed.

Each of these photocells works the same way as in the previous case, with the difference that a registered transition of the position by one photocell, has to be confirmed by the other photocell, to make sure that this transition is due to a movement of the disc.

To exclude that the two photocells are simultaneously in front of ZoneI, the distance between the two photocells must be:

$$n * \gamma + \gamma/2, \text{ with } \gamma = \frac{2\Pi}{2n} \text{ and } n > 0,$$

this distance is called "shift" in the PVS Theories. This distance ensures that if one photocell is in front of a ZoneI, then the other photocell is in the middle of a sector, transparent or opaque.

For instance, if photocell1 registers a transition from a transparent section (position1(V)) to an opaque one (position1(V)), photocell2 has to register the same transition before photocell1 can register once again the same transition (expressed differently: before photocell1 has registered two other transitions).

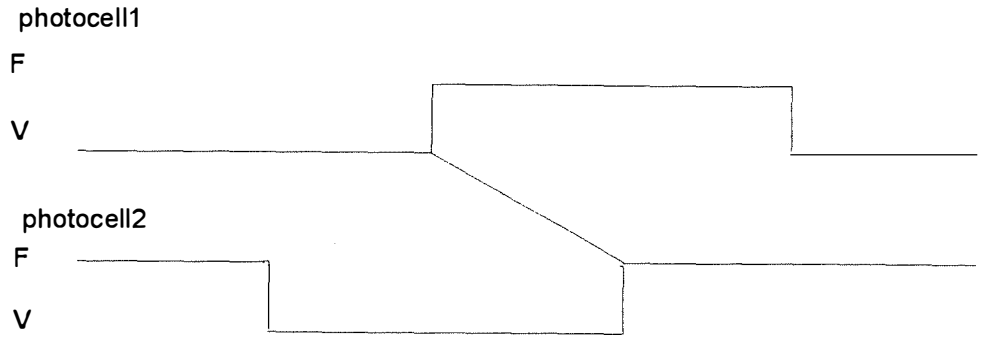

Figure 6.2: A confirmed transition (1)

The example of figure 6.2 shows a confirmed transition of the position. The photocell1 registers a transition and this is confirmed by the same transition from photocell2 and between these two transitions, there is no other transition.

The example of figure 6.3 shows that the first transition (I) of the position of photocell1 is not confirmed by photocell2, meanwhile photocell1 registers the same transition again (II) and this one is confirmed, in this example, by photocell2.

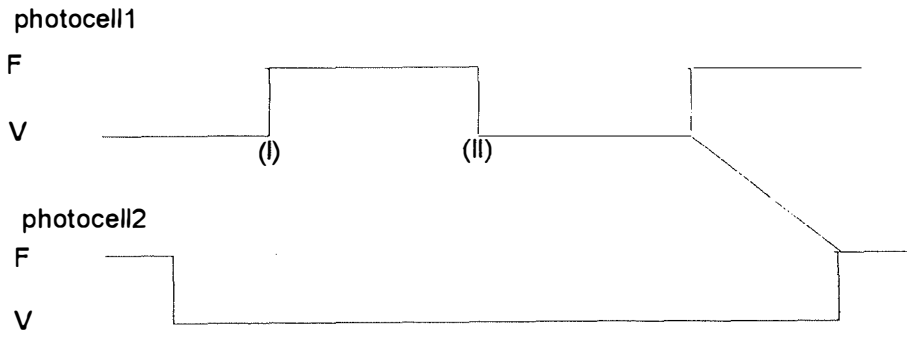

Figure 6.3: A confirmed transition (2)

6.3.2 The verification

The goal of the verification is to prove that there are only two possibilities to have a quantum and that the system cannot register a quantum in another situation.

The first possibility is an "Up" (a transition from a transparent section to an opaque one) of one photocell followed by an "Up" by the other one, without any other "Up" occurring between the two: a confirmed "Up". The second possibility is a confirmed "Down" (a transition from an opaque section to a transparent one).

Further more, we prove that between two registered quantums, the rotation of the disc is between $\gamma - \delta$ and $\gamma + \delta$.

In figure 6.4,

(1) is the earliest moment, to detect the 1st quantum

(2) is the latest moment, to detect the 1st quantum

(3) is the earliest moment, to detect the 2nd quantum

(4) is the latest moment, to detect the 2nd quantum

The smallest rotation is, when the 1st quantum is detected as late as possible (2) and the 2nd one as early as possible (3). The rotation of the disc between (2) and(3) is: $(2\gamma - \delta/2) - (\gamma + \delta/2) = \gamma - \delta$.

The biggest rotation is encountered, when the 1st quantum is detected as early as possible (1) and the 2nd one as late as possible (4). The rotation of the disc between (1) and (4) is: $(2\gamma + \delta/2) - (\gamma - \delta/2) = \gamma + \delta$.

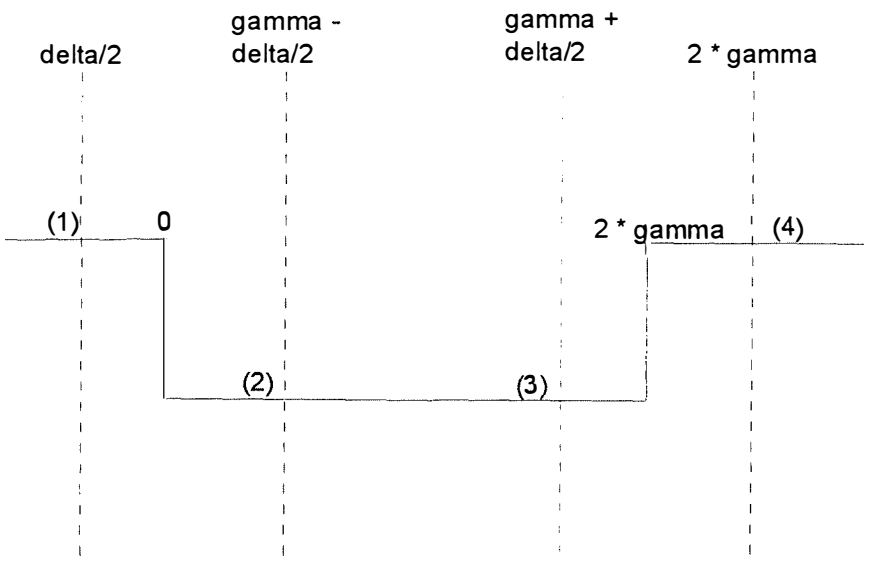

Figure 6.4: The rotation of the disc between two quantums

6.3.3 The proofs in PVS

The proof that the Energy_Meter with two photocells is correct and reliable is devided into two parts. First we prove that all the quantums are detected if the mechanism of the Energy_Meter is such that "it is not possible that the disc rotates in such a high speed, that between two activations, the angle of the disc changes more than $\gamma - \delta$". Secondly we demonstrate that all the counted quantums are due to energy consumption.

The rotation of the disc The rotation of the disc between two registered quantum is stated by "T1" Theorem:

```
T1: THEOREM
   Alw((Futr((ZoneF1 & TD_alpha(alpha1) & quantum), ti)
   & (Futr(ZoneV1, ti) & Futr(TD_alpha(alpha1), ti)
     & NextTime(quantum, ti)))
   => (old_pos + (gamma - delta) <= new_pos)
      & (old_pos + (gamma + delta) >= new_pos))
```

The general strategy of the proof of "T1" is shown in figure 6.5.

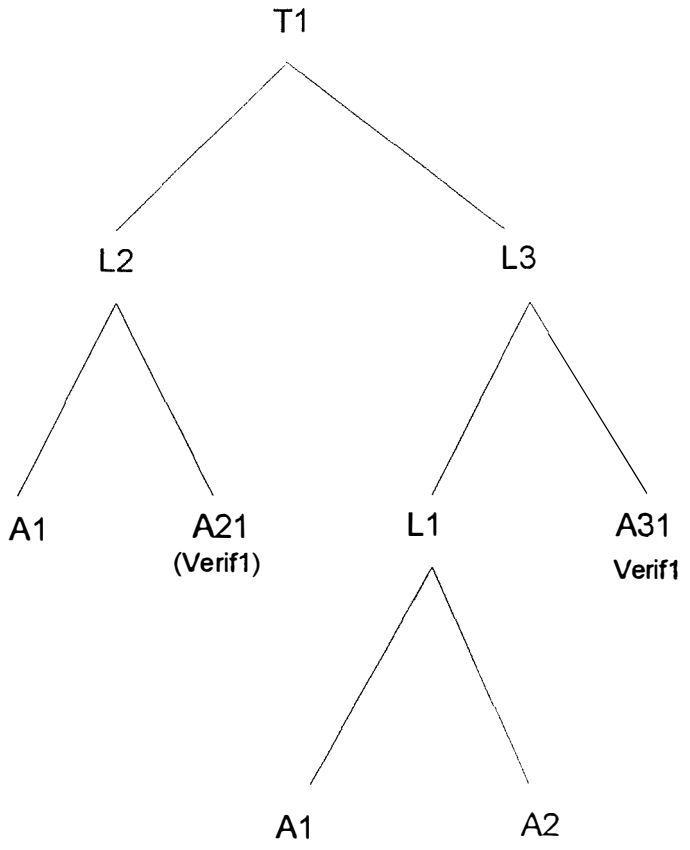

Figure 6.5: The proof-tree of the rotation-theorem

To prove this theorem, we first define an "old_pos" which correspond to the position of the 'first' quantum. This is done by the "A1" axiom of the "Verif2" Theory:

```
A1: AXIOM
   Alw(quantum & TD_alpha(alpha1)
   => (old_pos = alpha1))
```

The position of the 'second' quantum is the position of the disc, when the system has detected the next quantum and this position is called "new_pos".

```
L1: LEMMA
   Alw(NextTime(quantum, ti) & Futr(TD_alpha(alpha1), ti)
   => (new_pos = alpha1))
```

To proves this lemma in PVS, we need the following axiom, which states that a quantum is necessarily followed by an other quantum.

```
A2: AXIOM
   Alw(quantum & TD_alpha(alpha1)
   <=> (NextTime(quantum, ti) & Lasts(NOT quantum, ti))
       & ti > 0 & Futr(TD_alpha(alpha1), ti))
```

To be able to compare "old_pos" and "new_pos" we have to instantiate them. We look at the case where the first quantum is detected by a confirmed "Up" and the second one by a confirmed "down". This is done as follows:

```
L2: LEMMA
   Alw((ZoneF1 & TD_alpha(alpha1) & quantum)
   => (EXISTS n:(2*n*gamma + delta/2 < old_pos)
         & (old_pos < 2*n*gamma + gamma - delta/2))
      & NOT(ZoneI1 OR ZoneV1))

L3: LEMMA
   Alw((Futr(ZoneV1, ti) & Futr(TD_alpha(alpha1), ti)
         & NextTime(quantum, ti))
   => (EXISTS n:(2*n*gamma + gamma + delta/2 < new_pos)
         & (new_pos < 2*n*gamma + 2*gamma - delta/2))
      & Futr(NOT(ZoneI1 OR ZoneF1), ti))
```

The other case, where the first quantum is detected by a confirmed "Down" and the second one by a confirmed "Up", is similar.

The Theorem "T1" allows us to say that if the architecture of the Energy_Meter is such that "it is not possible that the disc rotates in such a high speed, that between two activations, the angle of the disc changes about more than $\gamma - \delta$", all the quantums are registered.

Further, we can consider the 'value' of the position obtained at one activation is valuable until the next activation because, the previous hypothesis says that it is not possible that the 'value' of position changes twice between two activations. This takes us from a punctual value of the position to a continuous one.

```
Alw(position1(V) & Past(position1(F), t1)
   & Past(position1(F), 2*t1)
=> (position1(V)
   & Since(NOT position1(V), position1(F))))
```

At time $-2*t_1$ and $-t_1$ a position1(F) is detected and now a position1(V), so we can conclude that up to now the position was always position1(F) as shown on figure 6.6.

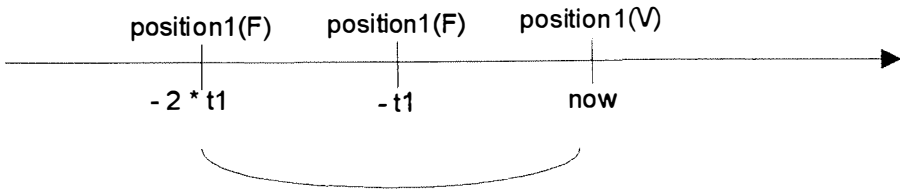

Figure 6.6: The 'value' of the position in time

A lot of axioms, built in the same way, are needed to be able to prove the different theorems and properties.

The counted quantums The Theorem "Quantum" states that the system counts only quantums, which are due to energy consumption.

```
Quantum: THEOREM
    Alw(
    (((position2(F)
          & Since(NOT position2(F), position2(V)))
    & Since (NOT
      (position2(F)
          & Since(NOT position2(F), position2(V))),
      (position1(F)
          & Since(NOT position1(F), position1(V)))))
    OR
    ((position2(V)
          & Since(NOT position2(V), position2(F)))
```

```
& Since (NOT
  (position2(V)
       & Since(NOT position2(V), position2(F))),
  (position1(V)
       & Since(NOT position1(V), position1(F))))))
<=> quantum)
```

The proof-tree of figure 6.7 shows us exactly which axioms are needed to prove the previous theorem.

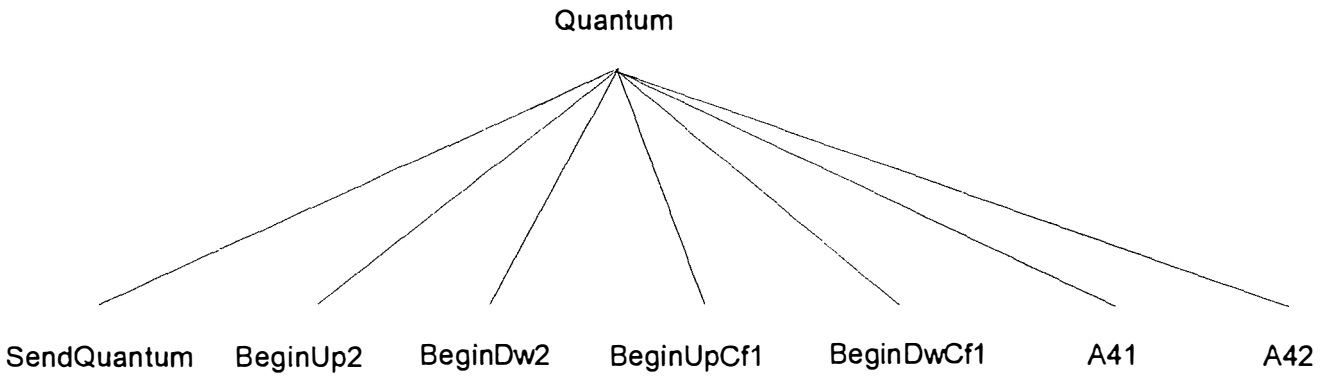

Figure 6.7: The proof-tree of the quantum-theorem

The axioms "SendQuantum", "BeginUp2", "BeginDw2", "BeginUpCf1" and "BeginDwCf1" are extracted from the "Detector_II" Theory.

As explained above, we need the following axioms to be able to convert the punctual value of the position to a continuous one:

```
A41: AXIOM
    Alw(Since(NOT bup2, bup1) <=>
    Since(NOT(position2(F)
        & Since(NOT position2(F), position2(V))),
        (position1(F)
        & Since(NOT position1(F), position1(V)))))
```

```
A42: AXIOM
    Alw(Since(NOT bdw2, bdw1) <=>
    Since(NOT(position2(V)
        & Since(NOT position2(V), position2(F))),
        (position1(V)
        & Since(NOT position1(V), position1(F)))))
```

6.3.4 The properties of the case with two photocells

After having shown, that the system count only a quantum in two different possibilities and that the rotation between two registered quantums is in the range $\gamma - \delta$, $\gamma + \delta$, we prove different properties of the Energy_Meter with two photocells. These properties correspond to the different situations the system can be faced to.

1. It is evident that if there is no transition of the positions, the system should not find a quantum.

   A.

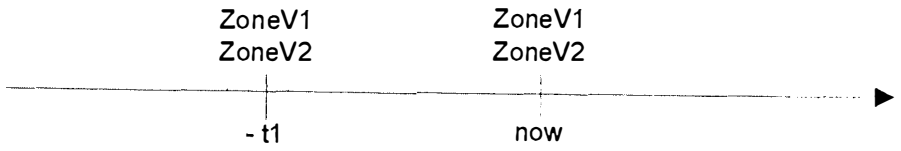

Figure 6.8: 1st property - version 1

```
P11: THEOREM
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   => NOT quantum)
```

B.

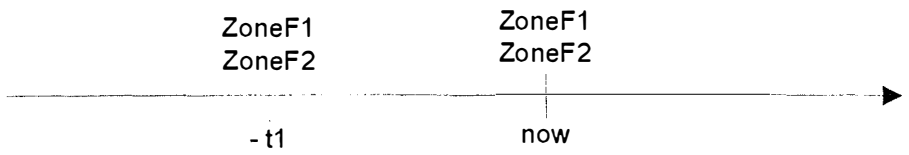

Figure 6.9: 1st property - version 2

```
P12: THEOREM
   Alw(ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   => NOT quantum)
```

2. A confirmed transition of the position is the situation in which the system should register a quantum.

   A.

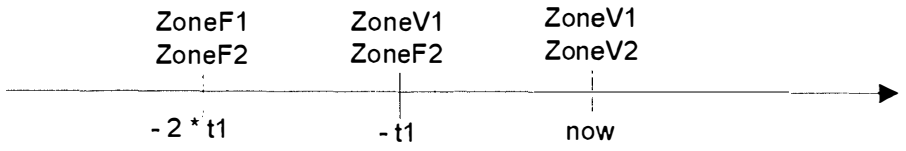

Figure 6.10: 2nd property - version 1

```
P21: THEOREM
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
```

```
   & Past((ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
   => quantum)
```

B.

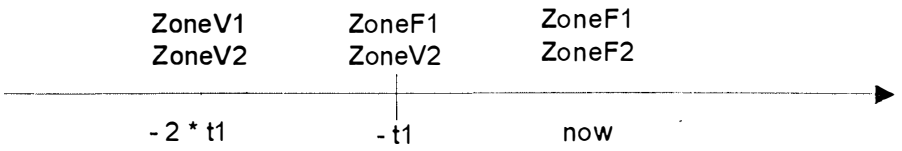

Figure 6.11: 2nd property - version 2

```
P22: THEOREM
   Alw(ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneF1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   & Past((ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
   => quantum)
```

3. A non confirmed transition of the position is not considered as a quantum by the system

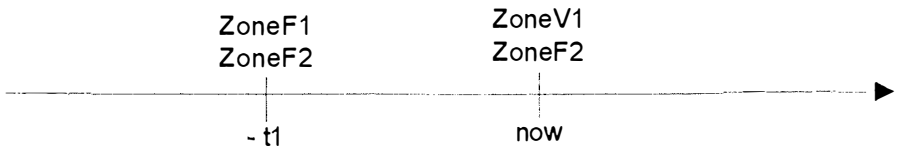

Figure 6.12: 3rd property

```
P3: THEOREM
   Alw(ZoneV1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   => NOT quantum)
```

4. If an activation occurs when one of the photocells is in front of a ZoneI, the system should only register ONE quantum. The "OR" result of the ZoneI should provide the result: (quantum XOR Past(quantum, t1))

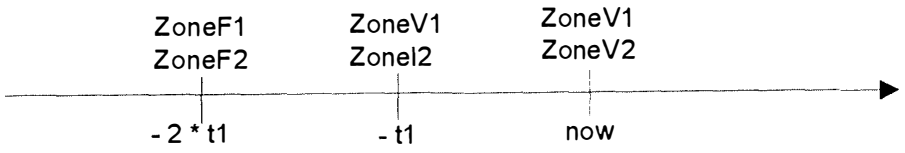

Figure 6.13: 4th property

A.

```
P41: THEOREM
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneI2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   & Past((ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
   => quantum OR Past(quantum, t1))
```

B.

```
P42: THEOREM
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneI2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
```

```
    & Past((ZoneF1 & ZoneF2 & activation
       & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
    => NOT (quantum & Past(quantum, t1)))
```

The proofs of the properties in PVS To prove these properties in PVS, we need to introduce a few new lemmas and axioms.

First of all, we have to consider the following lemmas. They are derived from the axioms of the "Detector_2" Theory. The axioms of the "Detector_2" Theory are defining the situations, where a quantum has to be detected and the following lemmas are stating the situations in which NO quantum is detected.

- ```
  L1: LEMMA
      Alw(NOT (bupconf1 OR bdwconf1) => NOT quantum)
  ```

  "bupconf1" stands for a confirmed "Up" (a transition from a transparent sector to an opaque one) and "bdwconf1" for a confirmed "Down" (a transition from an opaque sector to a transparent one). L1 states that if there is no confirmed "Up" and no confirmed "Down" then there is no quantum detected.

- ```
  L2: LEMMA
      Alw(NOT bup2 OR NOT Since(NOT bup2, bup1)
      => NOT bupconf1)
  ```

  "bup2" stands for an "Up" of photocell1 and 'bup1' for an "Up" of photocell2. L2 states that if an "Up" of photocell1 is not followed by an "Up" of photocell2 then this "Up" is not confirmed.

- ```
  L3: LEMMA
      Alw(NOT bdw2 OR NOT Since(NOT bdw2, bdw1)
      => NOT bdwconf1)
  ```

  "bdw2" stands for an "Down" of photocell1 and 'bdw1' for an "Down" of photocell2. L3 states that if an "Down" of photocell1 is not followed by an "Down" of photocell2 then this "Down" is not confirmed.

- ```
  L4: LEMMA
      Alw(NOT position1(F)
          OR NOT Since(NOT position1(F), position1(V))
      => NOT bup1)
  ```

L4 states that if there is no position1(F) or up to now no position1(V) than there is no "Up" of photocell1.

- ```
  L5: LEMMA
     Alw(NOT position2(F)
         OR NOT Since(NOT position2(F), position2(V))
     => NOT bup2)
  ```

  L5 states that if there is no position2(F) or up to now no position2(V) than there is no "Up" of photocell2.

- ```
  L6: LEMMA
     Alw(NOT position1(V)
         OR NOT Since(NOT position1(V), position1(F))
     => NOT bdw1)
  ```

  L6 states that if there is no position1(V) or up to now no position1(F) than there is no "Down" of photocell1.

- ```
  L7: LEMMA
     Alw(NOT position2(V)
         OR NOT Since(NOT position2(V), position2(F))
     => NOT bdw2)
  ```

  L7 states that if there is no position2(V) or up to now no position2(F) than there is no "Down" of photocell2.

In the TRIO+ specification the type of position is a predicate, but in PVS we had to define position as a TD_Var[Time, Pos].TD_var, to consider that the predicate position is time dependent.

The Td_Var[Time, Pos].TD_var evaluates only if e.g. position1(F) is true or false. We know that if position1(F) is true then position1(V) has to be false, but PVS evaluates position1(F) and position1(V) as two independent predicates and does not establish the link. The following axioms are creating this link explicitely:

```
A1F: AXIOM
   Alw(NOT position1(F) <=> position1(V))

A1V: AXIOM
   Alw(NOT position1(V) <=> position1(F))
```
```

```
A1F: AXIOM
   Alw(NOT position2(F) <=> position2(V))

A1F: AXIOM
   Alw(NOT position2(V) <=> position2(F))

A2Dw: AXIOM
   Alw(position2(V) & Past(position2(V), t1)
   => NOT Since(NOT position2(V), position2(F)))

A2Up: AXIOM
   Alw(position2(F) & Past(position2(F), t1)
   => NOT Since(NOT position2(F), position2(V)))
```

The proof of each property is done in three steps.

In the first step, a lemma shows in its consequent which values are attributed to the position at the different time instances in consulting the given zones.

The second lemma transforms these punctual values in continuous ones. The lemma is proved by using axioms stated before as explained in the beginning of this chapter.

The third step gives a lemma, which shows that with the continuous values obtained during the second step, the system correctly registers a quantum or not, depending on the property that is to be proved.

As all the properties are proved in the same way, we give only one example, the fourth property, to show how this method is applied. The fourth property consists in saying that the "OR" result of the ZoneI is the same as quantum XOR Past(quantum, t1). To prove it, we demonstrate, that the system gives us a quantum OR Past(quantum, t1) and NOT(quantum & Past(quantum, t1):

```
P41: THEOREM
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneI2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   & Past((ZoneF1 & ZoneF2 & activation
```

```
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
   => quantum OR Past(quantum, t1))

P42: THEOREM
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneI2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   & Past((ZoneF1 & ZoneF2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
   => NOT(quantum & Past(quantum, t1)))
```

The general structure of the proof of these two theorems is represented respectively on figure 6.14 and on figure 6.15.

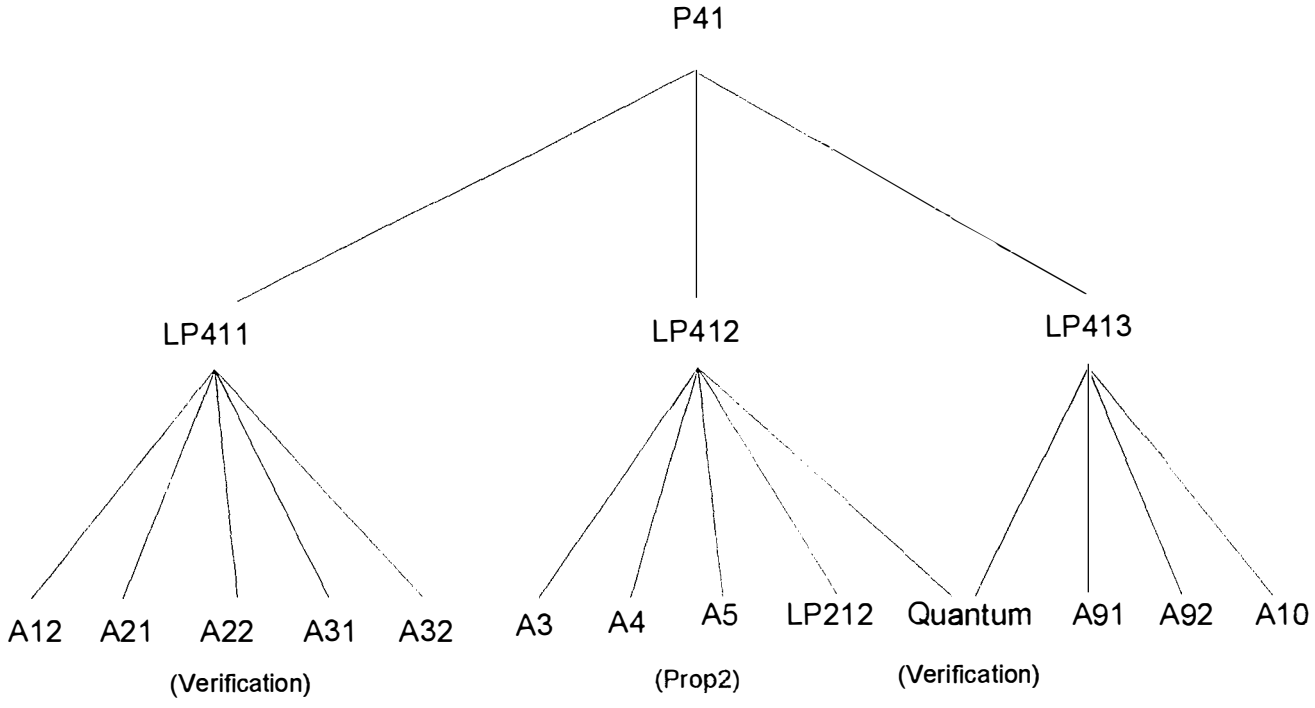

Figure 6.14: The proof-tree of the first part of the 4th property

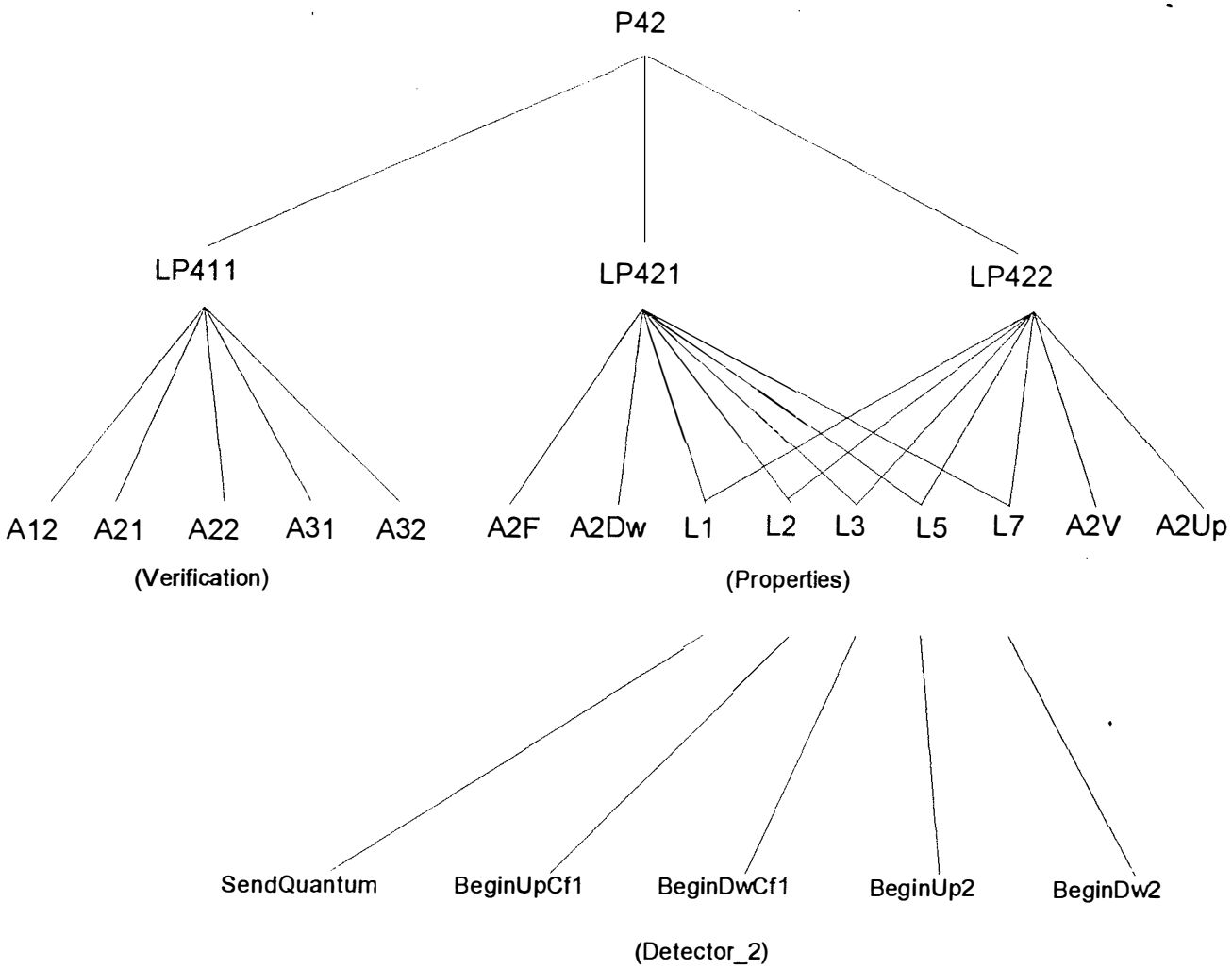

Figure 6.15: The proof-tree of the second part of the 4th property

Applying the method explained earlier, "P41" Theorem is proved in three steps. The "LP411" Lemma is stating the different positions which are obtained in the given situation.

```
LP411: LEMMA
   Alw(ZoneV1 & ZoneV2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)
   & Past((ZoneV1 & ZoneI2 & activation
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
   & Past((ZoneF1 & ZoneF2 & activation
```

```
      & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
   => position1(V) & position2(V)
      & Past((position1(V)
      & (position2(V) OR position2(F))), t1)
      & Past((position1(F) & position2(F)), t1 + t1))
```

Because of the OR result in the previous lemma, we have two possible cases, one of the cases is considered by the "LP412" Lemma

```
LP412: LEMMA
   Alw(position1(V) & position2(V)
   & Past((position1(V) & position2(F)), t1)
   & Past((position1(F) & position2(F)), t1 + t1)
   => quantum)
```

and the second one by the "LP413" Lemma.

```
LP413: LEMMA
   Alw(position1(V) & position2(V)
   & Past((position1(V) & position2(V)), t1)
   & Past((position1(F) & position2(F)), t1 + t1)
   => Past(quantum, t1))
```

These three lemmas allow us to prove the "P41" Theorem.

The second part of the property, is proved in a similar way.

The first step is exactly the same, so we reuse the lemma "LP411" to state the different positions which are obtained in the given situation.

Because of the NOT(...AND...) structure that should be proved and the OR result in the lemma "LP411", we have to prove that we can NOT have a detected quantum during the last activation AND one during the present activation.

The lemma "LP421" states the situation in which we have no detected quantum

```
LP421: LEMMA
   Alw(position1(V) & position2(V)
   & Past((position1(V) & position2(V)), t1)
   & Past((position1(F) & position2(F)), t1 + t1)
   => NOT quantum)
```

and lemma "LP422" states the situation in which we have no detected quantum during the last activation.

```
LP422: LEMMA
   Alw(position1(V) & position2(V)
   & Past((position1(V) & position2(F)), t1)
   & Past((position1(F) & position2(F)), t1 + t1)
   => NOT (Past(quantum, t1)))
```

These three lemmas allow us to prove the "P42" Theorem. As the theorems "P41" and "P42" are proved, the fourth property is proved as well.

Chapter 7

Conclusion

All the specifications in which the concept of time is used, can be devided into two sorts of requirements [15]:

- the use of relative time, this concept of time is only used to order events;
- the use of absolute time, this concept of time is used to define constraints between events, it is an explicite use of real time.

Our case study is a real-time system. It is a system in which "computerized control units obtain information about the environment of the system through sensors and are required to act on this environment within or after a bounded delay by commanding actuators" [16].

Often the real-time systems are used in critical systems and the safety feature is a crucial one. To us, this is one of the most important reason why it is important to demonstrate rigorously the crucial requirements of such a system [15, 16].

In chapter one, we introduce the specification language TRIO and we propose a specification of an Energy_Meter in TRIO+ in chapter four.

With the encoding of TRIO in PVS, we are able to verify the case study. In chapter six, we show how to prove that the Energy_Meter with one photocell is not rigorous enough and how to demonstrate that the Energy_Meter with two photocells ensures all the required qualities of the system.

During our stay in Milan, we realized that it can be rather difficult, by using the Proof Checker of PVS, to find the right way to prove a theorem.

But to demonstrate rigorously a specification, human review and inspection is not sufficient. A theorem prover, as the Proof Checker of PVS is very

useful for the verification of a specification and more particularly to prove boundary conditions. "It is difficult without complete formal verification support to completely formalize all necessary assumption - particularly at boundary conditions" [7]. The verification of the Energy_Meter with **one** photocell fails also at a boundary condition.

ALBERT is, as well as TRIO, a specification language which offers the possibility to specify real-time systems. So, it would be interesting, that ALBERT could also offer an verification tool to its users. This can be realized in encoding ALBERT-CORE in PVS.

In the case of TRIO/PVS, even if the user works at a low level of semantic encoding, it is rather easy to extract higher level proofs which are understandable [7]. As ALBERT-CORE provides the semantic foundation of ALBERT [5], the use of the encoding of ALBERT-CORE can be very similar to the use of the semantic encoding of TRIO and it could be possible that the structure of the different proofs of this work could be reused for doing similar proofs in the encoded semantic of ALBERT.

Bibliography

[1] A. Alborghetti, *Analisi di proprieta' di sistemi in tempo reale: Un approccio deduttivao basato sulla codifica della logica TRIO in PVS*, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 1996

[2] A. Borgida, J. Mylopoulos, R. Reiter, *One the Frame Problem in Procedure Specification*, IEEE Transactions on Software Engineering, 1995

[3] R. W. Butler, *An elementary tutorial on formal specification and verification using PVS*, NASA, Langley Research Center, 1993

[4] F. Chabot, *Semantic embedding of Albert-CORE within PVS* in Proceedings of the Doctoral Consortium of the third IEEE Symposium on Requirements Engineering, 1997

[5] P. Du Bois, *The AlbertII Language, On the Design and the Use of a Formal Specification Language for Requirements Analysis* PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, 1995

[6] E. Dubois, P. Du Bois, M. Petit, S. Wu, *Towards a Formal Agent-Oriented Definition of Security Requirements for Information Systems*, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, 1994

[7] R. D. Jeffords, *An Approach to Encoding the TRIO Logic in PVS*, Naval Research Laboratory, 1995

[8] D. Mandrioli, A. Morzenti, P. San Pietro, E. Crivelli, *Specification and Verification of real-time Systems in a logic framework: The TRIO environment*, Politecnico di Milano, Dipartimento di Elettronica e Informazione

[9] D. Mandrioli, *Applying Research Results in the industrial Environment: The Case of TRIO Specification Language*, Politecnico di Milano, Dipartimento di Elettronica e Informazione

[10] A. Morzenti, P. San Pietro, *Object Oriented Logical Specification of Time Critical Systems*, Politecnico di Milano, Dipartimento di Elettronica e Informazione

[11] A. Morzenti, M. Paci, F. Veroni, *Specifica TRIO+ dell' Unita' di Elaborazione Periferica*, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 1993

[12] S. Owre, N. Shankar, J.M. Rushby, *The PVS Specification Language (Beta Release)*, SRI International, Computer Science Laboratory, 1993

[13] S. Owre, N. Shankar, J.M. Rushby, *A Tutorial on Specification and Verification using PVS (Beta Release)*, SRI International, Computer Science Laboratory, 1993

[14] S. Owre, N. Shankar, J.M. Rushby, *The PVS Proof Checker: A Reference Manual (Beta Release)*, SRI International, Computer Science Laboratory, 1993

[15] N. Shankar, *Mechanized verification of real-time systems using PVS* (chapter 1)

[16] J.V. Skakkebaek, *A Verification Assistant for a real-time Logic* (chapter 1 - 4)

[17] *An encoding of TRIO in PVS*, Politecnico di Milano, Dipartimento di Elettronica e Informazione

[18] *AlbertII Abstract Syntax*, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, 1996

Appendix A

The proof-strategies in PVS

In chapter 2, we explain the proof of the Airline Reservation System. In this chapter, we intend to show how the different theorems are proved in PVS. Each of the following sections, represents all the proof steps of a particular theorem, which are saved in a .prf file and are re-executable.

A.1 Cancel_assn_inv Theorem

```
(|cancel_assn_inv| "" (SKOLEM!)
 (("" (FLATTEN)
   (("" (EXPAND "cancel_assn")
     (("" (ASSERT)
       (("" (EXPAND "assn_invariant")
         (("" (EXPAND "existence")
           (("" (EXPAND "uniqueness")
             (("" (EXPAND "member")
               (("" (FLATTEN)
                 (("" (SPLIT 1)
                   (("1" (SKOLEM! 1)
                     (("1" (CASE "flt!1 = flt!2")
                       (("1" (ASSERT)
                         (("1" (INST -2 "a!1" "flt!1")
                           (("1" (FLATTEN) (("1" (ASSERT) NIL)))))))
                        ("2" (ASSERT)
                         (("2" (INST -1 "a!1" "flt!2") NIL)))))))))))))))))))
```

```
                ("2" (SKOLEM! 1)
                 (("2" (CASE "flt!1 = flt!2")
                   (("1" (ASSERT)
                     (("1" (INST -3 "a!1" "b!1" "flt!1")
                       (("1" (FLATTEN) (("1" (ASSERT) NIL)))))))
                    ("2" (ASSERT)
                     (("2" (INST -2 "a!1" "b!1" "flt!2")
                       NIL)))))))))))))))))))))))))))))
```

A.2 MAe Theorem

```
(|MAe| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "existence")
     (("" (EXPAND "make_assn")
       (("" (AUTO-REWRITE! "member")
         (("" (SKOLEM! *)
           (("" (FLATTEN)
             (("" (INST -1 "a!1" "flt!2")
               (("" (ASSERT)
                 (("" (LIFT-IF -1)
                   (("" (SPLIT -1)
                     (("1" (GROUND)
                       (("1" (EXPAND "add")
                         (("1" (LEMMA "next_seat_ax")
                           (("1" (INST -1 "flt!1" "pref!1" "s1!1")
                             (("1" (GROUND) NIL)))))))))
                      ("2" (GROUND) NIL)))))))))))))))))))))
```

A.3 MAu Theorem

```
(|MAu| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "uniqueness")
     (("" (SKOSIMP)
       (("" (INST -1 "a!1" "b!1" "flt!2")
```

```
((""  (EXPAND "make_assn")
  (("" (EXPAND "member")
    (("" (ASSERT)
      (("" (LIFT-IF -1)
        (("" (LIFT-IF -2)
          (("" (GROUND)
            (("1" (EXPAND "pass_on_flight")
              (("1" (INST 2 "a!1")
                (("1" (INST 4 "b!1")
                  (("1" (EXPAND "add")
                    (("1" (EXPAND "member")
                      (("1" (ASSERT) NIL)))))))))))
             ("2" (EXPAND "pass_on_flight")
              (("2" (INST 2 "a!1")
                (("2" (INST 4 "b!1")
                  (("2" (EXPAND "add")
                    (("2" (EXPAND "member")
                      (("2" (ASSERT)
                        NIL))))))))))))))))))))
```

A.4 Make_assn_inv Theorem

```
(|make_assn_inv| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "assn_invariant")
     (("" (LEMMA "MAe")
       (("" (INST -1 "flt!1" "pas!1" "pref!1" "s1!1")
         (("" (LEMMA "MAu")
           (("" (INST -1 "flt!1" "pas!1" "pref!1" "s1!1")
             (("" (GROUND) NIL)))))))))))))
```

A.5 Initial_state_inv Theorem

```
(|initial_state_inv| "" (EXPAND "assn_invariant")
 (("" (EXPAND "initial_state")
   (("" (EXPAND "existence")
```

```
      (("" (EXPAND "uniqueness")
        (("" (EXPAND "emptyset")
          (("" (EXPAND "member") (("" (PROPAX) NIL))))))))))))
```

A.6 Make_cancel Theorem

```
(|make_cancel| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "pass_on_flight")
     (("" (EXPAND "cancel_assn")
       (("" (EXPAND "make_assn")
         (("" (EXPAND "member")
           (("" (LIFT-IF 2)
             (("" (SPLIT 2)
               (("1" (FLATTEN)
                 (("1" (APPLY-EXTENSIONALITY 1)
                   (("1" (HIDE 2)
                     (("1" (CASE "flt!1 = x!1")
                       (("1" (ASSERT)
                         (("1" (APPLY-EXTENSIONALITY 1)
                           (("1" (HIDE 2)
                             (("1" (REPLACE -1 * RL)
                               (("1" (INST 2 "x!2")
                                 (("1" (GROUND) NIL))))))))))))
                      ("2" (ASSERT) NIL)))))))))
                ("2" (GROUND)
                 (("2" (APPLY-EXTENSIONALITY 3)
                   (("2" (HIDE 4)
                     (("2" (CASE "flt!1 = x!1")
                       (("1" (ASSERT)
                         (("1" (APPLY-EXTENSIONALITY 1)
                           (("1" (EXPAND "add")
                             (("1" (HIDE 2)
                               (("1" (INST 4 "x!2")
                                 (("1" (IFF 1)
                                   (("1" (GROUND)
                                     (("1"
```

```
                                        (EXPAND "member")
                                        (("1" (ASSERT) NIL)))
                                       ("2"
                                        (EXPAND "member")
                                        (("2" (PROPAX) NIL)))
                                       ("3"
                                        (EXPAND "member")
                                        (("3" (ASSERT) NIL)))))))))))))))))
                          ("2" (ASSERT) NIL)))))))))))))))))))))))))))
```

A.7 Cancel_assn_inv Theorem

```
(|cancel_assn_inv| "" (SKOLEM!)
 (("" (FLATTEN)
   (("" (EXPAND "cancel_assn")
     (("" (EXPAND "assn_invariant")
       (("" (EXPAND "existence")
         (("" (EXPAND "uniqueness")
           (("" (EXPAND "one_per_seat")
             (("" (EXPAND "member")
               (("" (FLATTEN)
                 (("" (SPLIT 1)
                   (("1" (SKOLEM! 1)
                     (("1" (CASE "flt!1 = flt!2")
                       (("1" (ASSERT)
                         (("1" (INST -2 "a!1" "flt!1")
                           (("1" (FLATTEN) (("1" (ASSERT) NIL)))))))
                        ("2" (ASSERT)
                         (("2" (INST -1 "a!1" "flt!2") NIL)))))))
                    ("2" (SKOLEM! 1)
                     (("2" (CASE "flt!1 = flt!2")
                       (("1" (ASSERT)
                         (("1" (INST -3 "a!1" "b!1" "flt!1")
                           (("1" (FLATTEN) (("1" (ASSERT) NIL)))))))
                        ("2" (ASSERT)
                         (("2" (INST -2 "a!1" "b!1" "flt!2") NIL)))))))
                    ("3" (SKOLEM! 1)
```

```
                    (("3" (CASE "flt!1 = flt!2")
                      (("1" (ASSERT)
                        (("1" (INST -3 "a!1" "b!1" "flt!1")
                          (("1" (FLATTEN)
                            (("1" (ASSERT)
                              (("1" (INST -6 "a!1" "b!1" "flt!1")
                                (("1" (ASSERT) NIL)))))))))))
                       ("2" (ASSERT)
                        (("2" (INST -3 "a!1" "b!1" "flt!2")
                          NIL))))))))))))))))))))))))))))
```

A.8 MAs Theorem

```
(|MAs| "" (SKOLEM! *)
  (("" (FLATTEN)
    (("" (EXPAND "one_per_seat")
      (("" (EXPAND "make_assn")
        (("" (AUTO-REWRITE! "member")
          (("" (SKOLEM! *)
            (("" (FLATTEN)
              (("" (INST -1 "a!1" "b!1" "flt!2")
                (("" (ASSERT)
                  (("" (LIFT-IF -1)
                    (("" (LIFT-IF -2)
                      (("" (SPLIT -1)
                        (("1" (SPLIT -2)
                          (("1" (GROUND)
                            (("1" (EXPAND "add")
                              (("1" (HIDE -3)
                                (("1" (LEMMA "next_seat_ax_2")
                                  (("1"
                                    (INST -1 "flt!1" "pref!1" "s1!1" "b!1")
                                    (("1" (GROUND) NIL)))))))))
                             ("2" (EXPAND "add")
                              (("2" (HIDE -3)
                                (("2" (LEMMA "next_seat_ax_2")
                                  (("2"
```

```
                                    (INST -1 "flt!1" "pref!1" "s1!1" "a!1")
                                    (("2" (GROUND) NIL)))))))))))
                            ("2" (GROUND) NIL)))
                    ("2" (GROUND) NIL)))))))))))))))))))))))))))
```

A.9 Make_assn_inv Theorem

```
(|make_assn_inv| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "assn_invariant")
     (("" (LEMMA "MAe")
       (("" (INST -1 "flt!1" "pas!1" "pref!1" "s1!1")
         (("" (LEMMA "MAu")
           (("" (INST -1 "flt!1" "pas!1" "pref!1" "s1!1")
             (("" (LEMMA "MAs")
               (("" (INST -1 "flt!1" "pas!1" "pref!1" "s1!1")
                 (("" (GROUND) NIL)))))))))))))))))))
```

A.10 Initial_state_inv Theorem

```
(|initial_state_inv| "" (EXPAND "assn_invariant")
 (("" (EXPAND "initial_state")
   (("" (EXPAND "existence")
     (("" (EXPAND "uniqueness")
       (("" (EXPAND "one_per_seat")
         (("" (EXPAND "emptyset")
           (("" (EXPAND "member") (("" (PROPAX) NIL)))))))))))))))
```

A.11 Make_cancel Theorem

```
(|make_cancel| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "pass_on_flight")
     (("" (EXPAND "cancel_assn")
       (("" (EXPAND "make_assn")
         (("" (EXPAND "member")
```

```
(("" (LIFT-IF 2)
  (("" (SPLIT 2)
    (("1" (FLATTEN)
      (("1" (APPLY-EXTENSIONALITY 1)
        (("1" (HIDE 2)
          (("1" (CASE "flt!1 = x!1")
            (("1" (ASSERT)
              (("1" (APPLY-EXTENSIONALITY 1)
                (("1" (HIDE 2)
                  (("1" (REPLACE -1 * RL)
                    (("1" (INST 2 "x!2")
                      (("1" (GROUND) NIL)))))))))))
             ("2" (ASSERT) NIL)))))))))
     ("2" (GROUND)
      (("2" (APPLY-EXTENSIONALITY 3)
        (("2" (HIDE 4)
          (("2" (CASE "flt!1 = x!1")
            (("1" (APPLY-EXTENSIONALITY 1)
              (("1" (EXPAND "add")
                (("1" (HIDE 2)
                  (("1" (REPLACE -1 * RL)
                    (("1" (INST 4 "x!2")
                      (("1" (IFF 1)
                        (("1" (GROUND)
                          (("1"
                            (EXPAND "member")
                            (("1" (PROPAX) NIL)))
                           ("2"
                            (EXPAND "member")
                            (("2" (PROPAX) NIL)))
                           ("3"
                            (EXPAND "member")
                            (("3" (PROPAX) NIL)))))))))))))))))
           ("2" (ASSERT) NIL)))))))))))))
```

A.12 Cancel_inv_one_per_seat Theorem

```
(|cancel_inv_one_per_seat| "" (SKOLEM!)
 (("" (FLATTEN)
   (("" (EXPAND "cancel_assn")
     (("" (EXPAND "one_per_seat")
       (("" (EXPAND "member")
         (("" (SKOLEM!)
           (("" (CASE "flt!1 = flt!2")
             (("1" (ASSERT)
               (("1" (INST -2 "a!1" "b!1" "flt!2")
                 (("1" (FLATTEN) (("1" (ASSERT) NIL)))))))
              ("2" (ASSERT)
               (("2" (INST -1 "a!1" "b!1" "flt!2") NIL))))))))))))))))
```

A.13 Initial_one_per_seat Theorem

```
(|inital_one_per_seat| "" (EXPAND "one_per_seat")
 (("" (EXPAND "initial_state")
   (("" (EXPAND "emptyset") (("" (EXPAND "member") (("" (PROPAX) NIL)))))))
```

A.14 Make_putative Theorem

```
(|make_putative| "" (SKOLEM! *)
 (("" (FLATTEN)
   (("" (EXPAND "make_assn")
     (("" (LIFT-IF 2)
       (("" (GROUND)
         (("1" (EXPAND "pass_on_flight")
           (("1" (SKOLEM! -1) (("1" (INST 1 "a!1") (("1" (GROUND) NIL))))))
          ("2" (HIDE 4)
           (("2" (LEMMA "MAe")
             (("2" (INST -1 "flt!1" "pas!1" "pref!1" "s1!1")
               (("2" (GRIND) NIL)))))))))))))))))
```

A.15 Cancel_putative Theorem

```
(|cancel_putative| "" (SKOLEM!)
 (("" (SKOLEM! -1)
   (("" (EXPAND "cancel_assn")
     (("" (EXPAND "member") (("" (GROUND) NIL)))))))))
```

Appendix B

The TRIO+ specification

In this chapter, we represent the whole specification of the Energy_Meter Case Study in TRIO+.

At the end of this paragraph, 4 general classes are specified, which are inherited by different classes of the two specifications, of the Energy_Meter with one photocell and the one with two photocells.

In the first section, the specification of the Energy_Meter with one photocell is represented. For the specification of the Energy_Meter with two photocells, in the second section, only the classes which are different to those of the first specification, are represented.

Class ENERGY_METER
Visible m_power, disturb, med_power, inst_power,
total, post_prog, tariff_m_post,
d_pub_hol_post, t_pub_hol_post, d_hol_post, t_hol_post
TD Items
vars med_power, inst_power, disturb: real;
total: integer;
predicates tariff_m_post([1..31], [1..12], [1..48]);
post_prog
functions d_pub_hol_post([1..3]): [0..48]
hours of the updated weekly rest
t_pub_hol_post([1..3]): T1, T2, T3
tariff of the updated weekly rest
d_hol_post([1..5]): [0..48]
hours of an updated working day

t_hol_post([1..5]): T1. T2. T3
tariff of the updated working hours

Modules

optic_disc: OPTIC_DISC
g_ferraris: G_FERRARIS
detector: DETECTOR
tariff_program: TARIFF_PROGRAM
distributor: DISTRIBUTOR
totalizer: TOTALIZER
calendar: CALENDAR

Connections

{ g_ferraris.vibration	optic_disc.vibration
g_ferraris.ϕ	optic_disc.ϕ
m_power	g_ferraris.m_power
disturb	optic_disc.disturb
optic_disc.position	detector.position
detector.activation	optic_disc.activation
detector.quantum	distributor.quantum
calendar.day	tariff_program.day
calendar.month	tariff_program.month
calendar.hour	tariff_program.hour
post_prog	tariff_program.post_prog
tariff_m_post	tariff_program.tariff_m_post
d_pub_hol	tariff_program.d_pub_hol
t_pub_hol	tariff_program.t_pub_hol
d_hol	tariff_program.d_hol
t_hol	tariff_program.t_hol
tariff_program.fluent_tariff	distributor.fluent_tariff
distributor.med_power	med_power
distributor.inst_power	inst_power
distributor.energy_weight	totalizer.energy_weight
totalizer.total	total }

The connections represent the communication between the different modules.

end ENERGY_METER

Class SUM2TDVAR[VARTYPE]
general class of the addition of 2 variables of "varType"
Visible var1, var2, sum
TD Items
vars var1, var2, sum: varType
Axioms
sum = var1 + var2

end SUM2TDVAR

Class SUM3TDVAR[VARTYPE]
general class of the addition of 3 variables of "varType"
Visible var1, var2, var3, sum
TD Items
vars var1, var2, var3, sum: varType
Axioms
sum = var1 + var2 + var3

end SUM3TDVAR

Class DETECTOR
Visible position, activation, quantum
TI Items
consts δ_1, δ_2: real
TD Items
predicates position({F, V})
activation
quantum
confirm_transition
Axioms
Cycle:
Becomes(activation) $\rightarrow$
Lasts(activation, δ_1)
$\wedge$ Futr(NextTime(activation, δ_2), δ_1)
defines the cycle of activations

SendQuantum:
confirm_transition $\leftrightarrow$ quantum
a quantum is registered, when the system has detected a confirmed transition

end DETECTOR

B.1 The Energy_Meter with one photocell

Class ENERGY_METER_I
inherits ENERGY_METER **redefine** DETECTOR, OPTIC_DISC
Modules detector: DETECTOR_I
optic_disc: OPTIC_DISC_I

end ENERGY_METER_I

Class G_FERRARIS
Visible m_power, ϕ, vibration
TI Items
consts Φ_ϵ: real
TD Items
vars m_power, ϕ, vibration: real
Axioms
(Past (start, t) $\rightarrow$ vibration = A sen ω_m t)
Alw ($|$ vibration $| \leq \Phi_\epsilon$)
This axiom states that the vibration is an stationary oscillation.

end G_FERRARIS

Class SUM3TDREAL
is Sum3TdVar[Real]
end SUM3TDREAL

Class SUM3TDREAL
is Sum3TdVar[Real]
instantiation of Sum3TdVar
end SUM3TDREAL

Class PHOTO_CELL
Visible α, γ, activation, position
TI Items
consts δ, γ: real

TD Items

predicates position({F, V})
activation

vars α: real
indicates the disc position

Axioms

r: F, V

Iff_activation:

$$\exists r\ \text{position}(r) \leftrightarrow \text{activation}$$

the position of the disc is only evaluated, when the photocell is activated

Poss_lecture:

$$\frac{\delta}{2} \leq \frac{\gamma}{3}$$

this sufficient condition is guaranteeing that the Zone of Indecision is not too big

Lecture:

$$\text{activation} \rightarrow \left\{ \begin{array}{c} \left\{ \begin{array}{c} 0 \leq \alpha \bmod 2\gamma \leq \frac{\delta}{2} \\ \vee \\ \gamma - \frac{\delta}{2} \leq \alpha \bmod 2\gamma \leq \gamma + \frac{\delta}{2} \\ \vee \\ 2\gamma - \frac{\delta}{2} \leq \alpha \bmod 2\gamma \leq 2\gamma \end{array} \right\} \\ \rightarrow \{\text{position(F)} \vee \text{position(V)}\} \\ \wedge \\ (\frac{\delta}{2} < \alpha \bmod 2\gamma < \gamma - \frac{\delta}{2}) \\ \rightarrow \text{position(F)} \\ \wedge \\ (\gamma + \frac{\delta}{2} < \alpha \bmod 2\gamma < 2\gamma - \frac{\delta}{2}) \\ \rightarrow \text{position(V)} \end{array} \right\}$$

position(F) = opaque sector of the disc
position(V) = transparent sector of the disc
the value of the predicate "position" is stated in relation with the angle of the disc

end PHOTO_CELL

Class OPTIC_DISC_I
Visible disturb, position, activation, ϕ, vibration
TI Items
consts γ: real
n, N: integer
TD Items
predicates position({F, V})
activation
vars disturb: real
external cause of the system
vibration: real
intrinsic cause of the system
ϕ: real
position of the disc
ϕ_Ne_m: real
coeff. of the used Energy_Meter
Modules photo_cell: PHOTO_CELL
+3: SUM3RDREAL
Connections

{ disturb	+3.disturb
vibration	+3.vibration
ϕ_Ne_m	+3.ϕ_Ne_m
+3.α	photo_cell.α
γ	photo_cell.γ
activation	photo_cell.activation
photo_cell.position	position }

Axioms
PosDisc_incl:
$$\phi\text{\_Ne\_m} = \frac{\Phi}{N}$$
makes the position of the optic disc independent of the used Energy_Meter model

AngleOpaqueSector:
$$\gamma = \frac{2\Pi}{2n}$$
size of one opaque or transparent sector

NumOpaqueSector:

$$n = 10$$

the number of opaque sectors of the optic disc, the total number of sectors (opaque and transparent) is $2n$

end OPTIC_DISC_I

Class SHIFT_REGISTER
Visible position, content, insert
TI Items

vars content: all_1, all_0, mix
predicates position({F, V})
insert
slot([1..8])

Axioms

vars i: [1..8]
j: [1..7]

Initial_value:

$$\text{AlwP}(\neg \text{ insert}) \rightarrow \forall i \neg \text{ slot(i)}$$

the initial value of all the elements of the "slot" array is false

Postponement:

$$\text{insert} \rightarrow \left(\begin{array}{c} \text{position(F)} \rightarrow \text{Until\_ei(slot[8], insert)} \\ \wedge \text{position(V)} \rightarrow \text{Until\_ei}(\neg \text{slot[8], insert)} \\ \wedge \forall j \left(\begin{array}{c} \text{slot[j + 1]} \rightarrow \text{Until\_ei(slot[j], insert)} \\ \wedge \neg \text{slot[j + 1]} \rightarrow \text{Until\_ei}(\neg \text{slot[j], insert)} \end{array} \right) \end{array} \right)$$

if the predicate "insert" is true, then the update of the elements value of the "slot" array will be made and the new values will be memorized

Consistence:

$$(\forall \text{ i } (\neg \text{ insert} \wedge \text{slot[i]}) \rightarrow \text{slot[i]})$$

condition of consistence from one "insert" to the next one

Content_0:

$$(\forall\ \mathrm{i}\ \neg\ \mathrm{slot(i)}) \leftrightarrow (\mathrm{content} = \mathrm{all\_0})$$

if no element of the "slot" array is true, then "content" gets the value "all_0"

Content_1:

$$(\forall\ \mathrm{i}\ \mathrm{slot(i)}) \leftrightarrow (\mathrm{content} = \mathrm{all\_1})$$

if all the elements of the "slot" array are true, then "content" gets the value "all_1"

end SHIFT_REGISTER

Class DETECTOR_I
inherits DETECTOR
rename CONFIRM_TRANSITION **as** INSERT
redefine SENDQUANTUM
TI Items
consts δ: real
TD Items
vars content: all_1, all_2, mix
precState: 0, 1
Modules shift_register: SHIFT_REGISTER
Connections

{ content	shift_register.content
insert	shift_register.insert
position	shift_register.position }

Axioms

Const_0:

$$\delta = 20\ \mu\mathrm{sec}$$

defines the instant of sampling

Const_1:

$$\delta_1 = 25\ \mu\mathrm{sec}$$

defines the duration of the activation

Const_2:

$$\delta_2 = 3{,}1\ \mathrm{msec}$$

defines the interval between two activations

Insertion:
Becomes(activation) $\leftrightarrow$ Futr(insert,δ)
the sampling is registered $\delta = 20\mu sec$ *after the activation has started*

Transition1:
Becomes(content = all_1)
$\rightarrow$ Until$_w$_ei(precState = 1. content = all_0)
when "content" becomes "all_1", means that the previous state will have the value 1 *until the value of "content" changes to "all_0"*

Transition2:
Becomes(content = all_0)
$\rightarrow$ Until$_w$_ei(precState = 0. content = all_1)
when "content" becomes "all_0", means that the previous state will have the value 0 *until the value of "content" changes to "all_1"*

SendQuantum:
(Becomes(precState = 1) $\vee$ Becomes(precState = 0))
$\leftrightarrow$ quantum
a quantum is registered when a transition from a state "all_1" to a state "all_0" or vice-versa is detected

end DETECTOR_I

Class CALENDAR
Visible day, month, hour
TD Items
vars day[1..31]
month[1..12]
hour[1..48]
expressed in half hours

end CALENDAR

Class TARIFF_PROGRAM
Visible day, month, hour, post_prog, tariff_m_post,
d_pub_hol_post, t_pub_hol_post, d_hol_post, t_hol_post, fluent_tariff
TD Items
vars day[1..31]
month[1..12]
hour[1..48]
fluent_tariff: T1, T2, T3, T4
tariff of possible "mobile points"
f0, f1, f2, f3, f4, f5: [0..48]
working hours
F0, F1, F2, F3: [0..48]
hours of weekly rest
functions tariff([1..31, [1..12], [1..48]: T1, T2, T3;
tariff calculated relatively to the fixed period of time
d_pub_hol_post([1..3]): [0..48];
hours of the updated weekly rest
d_pub_hol([1..3]): [0..48],
hours of the weekly rest
t_pub_hol_post([1..3]): T1, T2, T3;
tariff of the updated weekly rest
t_pub_hol([1..3]): T1, T2, T3;
tariff of the weekly rest
d_hol_post([1..5]): [0..48];
hours of an updated working day
d_hol([1..5]): [0..48],
hours of a working day
t_hol_post([1..5]): T1, T2, T3;
tariff of the updated working time
t_hol([1..5]): T1, T2, T3;
tariff of the working time
predicates tariff_m([1..31], [1..12], [1..48]);
tariff_m_post([1..31], [1..12], [1..48]);
public_holiday([1..31], [1..12]);
post_prog

Axioms

vars t: T1, T2, T3, T4
f: [1..3]
d, h: [0..48]
g: [1..31]
m: [1..12]

Mobil_pt:
fluent_tariff = T4 ↔ tariff_m(day, month, hour)
"fluent_tariff" is a tariff of a mobile point if the predicate "tariff_m" is true for the current date (day, month and hour).

Fix:
¬ tariff_m(day, month, hour)
→ fluent_tariff = tariff(day, month. hour)
If the predicate "taruff_m" is false for the current date (day, month and hour) then the "fluent_tariff" is furnished by the function "tariff" to which the current date (day, month and hour) is applied.

Public_holiday:
public_holiday(day, month)
→ d_pub_hol(1) + d_pub_hol(2) + d_pub_hol(3) = 48
If the predicate "public_holiday" is true for the current date (day, month and hour) then the sum of the different 'hours of the weekly rest' is a total day expressed in half hours (48).

Working:
¬ public_holiday(day, month)
→ d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4) + d_hol(5) = 48
If the predicate "public_holiday" is false for the current date (day, month and hour) then the sum of the different 'working hours' is a total day expressed in half hours (48).

Ext_hol_0:
f0 = 0
defines the first foreseeable period of hours of a working day

Ext_hol_1:
f1 = d_hol(1)

Ext_hol_2:
f2 = d_hol(1) + d_hol(2)

Ext_hol_3:

$$f3 = d\_hol(1) + d\_hol(2) + d\_hol(3)$$

Ext_hol_4:

$$f4 = d\_hol(1) + d\_hol(2) + d\_hol(3) + d\_hol(4)$$

Ext_hol_5:

$$f5 = d\_hol(1) + d\_hol(2) + d\_hol(3) + d\_hol(4) + d\_hol(5)$$

Ext_pub_hol_0:

$$F0 = 0$$

Ext_pub_hol_1:

defines the first foreseeable period of hours of a rest day

$$F1 = d\_pub\_hol(1)$$

Ext_pub_hol_2:

$$F2 = d\_pub\_hol(1) + d\_pub\_hol(2)$$

Ext_pub_hol_3:

$$F3 = d\_pub\_hol(1) + d\_pub\_hol(2) + d\_pub\_hol(3)$$

Tariffs:

$$\text{tariff(day, month, hour)} = t \leftrightarrow$$

$$\left(\begin{array}{c} \text{public\_holiday(day, month)} \\ \wedge \left(\begin{array}{c} ((F0 < \text{hour} < F1) \wedge (t\_pub\_hol(1) = t)) \\ \vee((F1 < \text{hour} < F2) \wedge (t\_pub\_hol(2) = t)) \\ \vee((F2 < \text{hour} < F3) \wedge (t\_pub\_hol(3) = t)) \end{array} \right) \\ \vee \\ \neg\text{public\_holiday(day, month)} \\ \wedge \left(\begin{array}{c} ((f0 < \text{hour} < f1) \wedge (t\_hol(1) = t)) \\ \vee((f1 < \text{hour} < f2) \wedge (t\_hol(2) = t)) \\ \vee((f2 < \text{hour} < f3) \wedge (t\_hol(3) = t)) \\ \vee((f3 < \text{hour} < f4) \wedge (t\_hol(4) = t)) \\ \vee((f4 < \text{hour} < f5) \wedge (t\_hol(5) = t)) \end{array} \right) \end{array} \right)$$

defines the current tariff (mobile points excluded) based on the current date and type of day, public rest or working day

Postponement:

$$\text{post\_prog} \rightarrow \left[\begin{array}{c} \forall f,\forall d,\forall t \\ \left(\begin{array}{c} (\text{d\_pub\_hol\_post(f)} = \text{d} \rightarrow \text{d\_pub\_hol(f)} = \text{d}) \\ \wedge \text{Until (d\_pub\_hol(f)} = \text{d, post\_prog)} \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} (\text{t\_pub\_hol\_post(f)} = \text{t} \rightarrow \text{t\_pub\_hol(f)} = \text{t}) \\ \wedge \text{Until (t\_pub\_hol(f)} = \text{t, post\_prog)} \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} (\text{d\_hol\_post(f)} = \text{d} \rightarrow \text{d\_hol(f)} = \text{d}) \\ \wedge \text{Until (d\_hol(f)} = \text{d, post\_prog)} \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} (\text{t\_hol\_post(f)} = \text{d} \rightarrow \text{t\_hol(f)} = \text{d}) \\ \wedge \text{Until (t\_hol(f)} = \text{d, post\_prog)} \end{array} \right) \end{array} \right] \wedge \left(\begin{array}{c} \forall g,\forall m,\forall h \\ \left(\begin{array}{c} (\text{tariff\_m\_post(g, m, h)} = \text{tariff\_m(g, m, h))} \\ \wedge \text{Until(tariff\_m(g, m, h), post\_prog)} \end{array} \right) \\ \wedge \\ \neg \left(\begin{array}{c} (\text{tariff\_m\_post(g, m, h)} = \neg\text{tariff\_m(g, m, h))} \\ \wedge \text{Until}(\neg\text{tariff\_m(g, m, h), post\_prog)} \end{array} \right) \end{array} \right)$$

The command to update the tariff program is sent by an external and produces a variation of the crucial datas of this program:
- the duration to each time segment for the working days as for the days of the weekly rest;
- the tariff of each time segment, possibly the tariff of the mobile point.
These datas will be replaced b the correspondent updated ones and will stay valid until the next update command.

end TARIFF_PROGRAM

Class DISTRIBUTOR
Visible quantum, fluent_tariff, energy_weight, med_power, inst_power
TD Items
vars fluent_tariff: T1, T2, T3, T4
nq: integer
consts k: real

predicats quantum
energy_weight (T1, T2, T3, T4)
ckPot
defines the interval of the power calculation

Axioms

vars t_1, t_2: T1, T2, T3, T4
t: real

Valid_quantum:
quantum $\rightarrow$ energy_weight(fluent_tariff)
The acquisition of a quantum furnishes the tariff which has to be applied to this quantum.

Non_valid_quantum:
$\neg$ quantum $\rightarrow \neg\exists t_1$ energy_weight(t_1)
The tariff which has to be applied to a quantum of consumed energy is not defined for the moments different to an acquisition of a quantum.

Unicity:
$\exists\ t_1, t_2$ (energy_weight(t_1) $\wedge$ energy_weight(t_2) $\rightarrow t_1 = t_2$)
The tariff which has to be applied to a quantum of energy is unique

Clock:
Sometimes(ckPot $\wedge$ Lasts($\neg$ ckPot, t))
$\wedge$ Always(ckPot $\leftrightarrow$ Futr(ckPot, t))
The clock defines the interval of power calculation; (does a tick at each Δ of time units).

Initial_Quantum:
ckPot $\wedge$ quantum $\rightarrow$ nq = 1
The initial value of "nq" at the beginning of an interval is 1 when at the moment of the initialization the predicate "quantum" is true.

Initial_Non_Quantum:
ckPot $\wedge\neg$ quantum $\rightarrow$ nq = 0
The initial value of "nq" at the beginning of an interval is 0 when at the moment of the initialization the predicate "quantum" is false.

Incrementation:

$$\text{quantum} \wedge \exists\, t\ (\text{LastTime}(\text{quantum} \vee \text{ckPot}, t) \wedge \text{Past}(nq = n, t)) \rightarrow nq = nq + 1$$

If a quantum 'arrives' then "nq" is incremented

Med_Power:

$$\text{ckPot} \wedge \exists\, t \left(\begin{array}{c} \text{LastTime}(\text{quantum} \vee \text{ckPot}, t) \\ \wedge \text{Past}(nq = n, t) \\ \rightarrow \text{med\_power} = km \\ \wedge \text{Until}(\text{med\_power} = km, \text{ckPot})) \end{array} \right)$$

If the predicate "ckPot" is true then "med_power is calculated and its value is valid until the predicate "ckPot" will be true for the next time.

Inst_power:

$$\text{quantum} \wedge \exists\, t\ (\text{LastTime}(\text{quantum}, t) \rightarrow \text{inst\_power} = \tfrac{k}{t})$$

"inst_power is calculated on the base of the time between a quantum and the previous one.

end DISTRIBUTOR

Class COUNTER
Visible add
TD Items
vars tot, n: integer
predicates add
Axioms
UpToNow(tot = n) ∧ add → Until_w_ie(tot = n + 1, add)
This axiom defines the incrementation of "tot".

end COUNTER

Class TOTALIZER
Visible energy_weight, total
TD Items
vars total, n: integer
predicates energy_weight(T1, T2, T3, T4)
Modules partial_tot: array[T1..T4] of COUNTER

Axioms

vars t, t_1, t_2: T1, T2, T3, T4

Tot:

UpToNow(total = n) $\wedge \exists t_1$ energy_weight(t_1)

$\rightarrow$ Until_ie (total = n + 1, $\exists t_2$ energy_weight(t_2))

In the moment, in which the tariff that has to be applied, is defined, the total of consumed energy quantums is incremented (+1). This value stays valid as long as the current tariff is valid.

Partial:

(energy_weight(t) $\leftrightarrow$ partial_tot[t].add)

The moment in which the predicate "energy_weight(t)" is true coincide with the moment in which the partial total of consumed energy, to which tariff "t" is applied, is incremented.

end TOTALIZER

B.2 The Energy_Meter with two photocells

Class ENERGY_METER_II
inherits ENERGY_METER **redefine** DETECTOR, OPTIC_DISC
Visible shift
TD Items

consts shift: real

Modules detector: DETECTOR_II

optic_disc: OPTIC_DISC_II

Connections

{ shift — optic_disc.shift
optic_disc.position1 — detector.position1
optic_disc.position2 — detector.position2 }

end ENERGY_METER_II

Class SUM2TDREAL
is Sum2TdVar[Real]
instantiation of Sum2TdVar
end SUM2TDREAL

Class SUM3TDREAL
is Sum3TdVar[Real]
instantiation of Sum3TdVar **end** SUM3TDREAL

Class PHOTO_CELL
Visible α, γ, activation, position
TI Items
consts δ, γ: real
TD Items
predicates position({F, V})
activation
vars α: real
indicates the disc position
Axioms
r: F, V

Iff_activation:
$\exists$r position(r) $\leftrightarrow$ activation
the position of the disc is only evaluated. when the photocell is activated

Poss_lecture:
$\frac{\delta}{2} \leq \frac{\gamma}{3}$
this sufficient condition is guaranteeing that the Zone of Indecision is not too big

Lecture:

activation $\rightarrow$

$$\left\{ \begin{array}{c} \left\{ \begin{array}{c} 0 \leq \alpha \mathrm{mod} 2\gamma \leq \frac{\delta}{2} \\ \vee \\ \gamma - \frac{\delta}{2} \leq \alpha \mathrm{mod} 2\gamma \leq \gamma + \frac{\delta}{2} \\ \vee \\ 2\gamma - \frac{\delta}{2} \leq \alpha \mathrm{mod} 2\gamma \leq 2\gamma \end{array} \right\} \\ \rightarrow \{\mathrm{position(F)} \vee \mathrm{position(V)}\} \\ \wedge \\ (\frac{\delta}{2} < \alpha \mathrm{mod} 2\gamma < \gamma - \frac{\delta}{2}) \\ \rightarrow \mathrm{position(F)} \\ \wedge \\ (\gamma + \frac{\delta}{2} < \alpha \mathrm{mod} 2\gamma < 2\gamma - \frac{\delta}{2}) \\ \rightarrow \mathrm{position(V)} \end{array} \right\}$$

position(F) = opaque sector of the disc
position(V) = transparent sector of the disc
the value of the predicate "position" is stated in relation with the angle of the disc

end PHOTO_CELL

Class OPTIC_DISC_II
inherits OPTIC_DISC_I
rename position **as** position1, photo_cell **as** cell_1
Visible shift, position2
TI Items
consts shift: real
shift represents the distance between the two photocells
TD Items
predicates position2({F, V})
Modules cell_2: PHOTO_CELL
+2: SUM3RDREAL
Connections

{ +3.sum	+2.var2
+2.sum	cell_2.α
shift	+2.var1
γ	cell_2.γ
cell_2.position	position2

activation cell_2.activation }

Axioms

Angelshift:

$$shift = \frac{\Phi}{2n}$$
$$shift = \frac{\gamma}{2} + n\gamma$$

end OPTIC_DISC_II

Class DETECTOR_II
inherits DETECTOR
rename POSITION **as** POSITION1, CONFIRM_TRANSITION **as** BUP1
redefine SENDQUANTUM
Visible position2
TD Items

predicates position({F, V})
bup1
begin of the rise of signal 1
bdw1
begin of the descend of signal 1
bup2
begin of the rise of signal 2
bdw2
begin of the descend of signal 2
bupconf1
begin of the confirmation of the rise of signal 1
bdwconf1
begin of the confirmation of the descend of signal 1

Axioms

BeginUp1:

$$bup1 \leftrightarrow position1(F) \wedge Since(\neg\, position1(F), position1(V))$$

"bup1" is true when photocell1 is in front of an opaque sector and when photocell1 had not been in front of an opaque sector since it had been in front of a transparent one

BeginDw1:

$$bdw1 \leftrightarrow position1(V) \wedge Since(\neg\, position1(V), position1(F))$$

"bdw1" is true when photocell1 is in front of a transparent sector and when photocell1 had not been in front of a transparent sector since it had been in front of an opaque one

BeginUp2:

$$\text{bup2} \leftrightarrow \text{position2(F)} \wedge \text{Since}(\neg\ \text{position2(F)},\ \text{position2(V)})$$

"bup2" is true when photocell2 is in front of an opaque sector and when photocell2 had not been in front of an opaque sector since it had been in front of a transparent one

BeginDw2:

$$\text{bdw2} \leftrightarrow \text{position2(V)} \wedge \text{Since}(\neg\ \text{position2(V)},\ \text{position2(F)})$$

"bdw2" is true when photocell2 is in front of a transparent sector and when photocell2 had not been in front of a transparent sector since it had been in front of an opaque one

BeginUpCf1:

$$\text{bupconf1} \leftrightarrow \text{bup2} \wedge \text{Since}(\neg\ \text{bup2},\ \text{bup1})$$

"bupconf1" is true when "bup2" is true and when "bup2" had not been true since "bup1" had been true

BeginDwCf1:

$$\text{bdwconf1} \leftrightarrow \text{bdw2} \wedge \text{Since}(\neg\ \text{bdw2},\ \text{bdw1})$$

"bdwconf1" is true when "bdw2" is true and when "bdw2" had not been true since "bdw1" had been true

SendQuantum:

$$\text{quantum} \leftrightarrow (\text{bupconf1} \vee \text{bdwconf1})$$

a quantum is only registered when a confirmed transition from an opaque to a transparent sector or vice-versa had been detected

end DETECTOR_II

Appendix C

The translation into TRIO/PVS

In this chapter, we show all the PVS files of the Case Study. In section C.1, all the files of the Energy_Meter with one photocell are represented and in section C.2 those of the Energy_Meter with two photocells.

C.1 The Energy_Meter with one photocell

First, we show the translation of the TRIO+ specification of the Energy_Meter with one photocell into TRIO/PVS. As explained in chapter 3, each TRIO+ class is represented in an PVS Theory.

The different theories are represented in section C.1.1. The PVS files related to the falsification are represented in section C.1.2. All the different lemmas and theorems are proved, but the different proof-steps are not represented in this document.

C.1.1 The specification in PVS

```
Energy_Meter: THEORY
BEGIN

    IMPORTING trio[int]
```

```
% Types
    Time: TYPE = int
    Pos: TYPE = {F,V}
    Month: TYPE = {n: posnat | 1  <= n & n <= 31} CONTAINING 1
    Day: TYPE = {n: posnat | 1 <= n & n <= 12} CONTAINING 1
    Hour: TYPE = {n: posnat | 1 <= n & n <= 48} CONTAINING 1
    Tarif: TYPE = {T1, T2, T3, T4}
    Hol1: TYPE = {n: posnat | 1 <= n & n <= 3} CONTAINING 1
    Hol2: TYPE = {n: posnat | 1 <= n & n <= 5} CONTAINING 1
% End types

    IMPORTING TD_Var[Time, Pos]
    IMPORTING TD_Var[Time, [Day, Month, Hour]]
    IMPORTING TD_Var[Time, [Day, Month]]
    IMPORTING TD_Var[Time, Tarif]
    IMPORTING TD_Var[Time, int]

% TD Vars
    position: TD_Var[Time, Pos].TD_var
    slot: TD_Var[Time, nat].TD_var
    tariff_m_post, tariff_m: TD_Var[Time, [Day, Month, Hour]].TD_var
    public_holiday: TD_Var[Time, [Day, Month]].TD_var
    energy_weight: TD_Var[Time, Tarif].TD_var

    TD_med_power: TD_Var[Time, real].TD_var
    TD_inst_power: TD_Var[Time, real].TD_var
    TD_disturb: TD_Var[Time, real].TD_var
    TD_phi: TD_Var[Time, real].TD_var
    TD_vibration: TD_Var[Time, real].TD_var
    TD_phi_Ne_m: TD_Var[Time, real].TD_var
    TD_alpha: TD_Var[Time, real].TD_var

    TD_total: TD_Var[Time, int].TD_var
% End TD Vars

    m_power: real
    post_prog: TD_Formula
```

```
    d_pub_hol_post: FUNCTION
        [Hol1 -> Hour]

    t_pub_hol_post: FUNCTION
        [Hol1 -> Tarif]

    d_hol_post: FUNCTION
        [Hol2 -> Hour]

    t_hol_post: FUNCTION
        [Hol2 -> Tarif]
END Energy_Meter

G_Ferraris: THEORY
BEGIN

    IMPORTING Energy_Meter

    phi_epsilon: real
    med_power, vibration: VAR real
    x, V, A, c: VAR real
    start: TD_Formula
    t: VAR Time

    Sinus: FUNCTION
        [real -> real]

    Sin1: AXIOM
        Alw ((Sinus(x) >= -1) &
        (Sinus(x) <= 1) &
        (Sinus(0) = 0))

    Vibr: AXIOM
        Alw(TD_med_power(med_power)
        & V = A * Sinus(c * med_power * t))

    Initial: AXIOM
        Alw(Past(start, t) => TD_vibration(vibration) & vibration = V)
```

```
    Initial_vibration: AXIOM
        Alw(TD_vibration(vibration) & (vibration) <= phi_epsilon)
END G_Ferraris

Optic_Disc: THEORY
BEGIN

    IMPORTING G_Ferraris

    Pi: real
    alpha, vibration, phi, phi_Ne_m, disturb: VAR real
    n, N, gamma: posnat

    Plus3: AXIOM
        Alw(TD_alpha(alpha) =>
        TD_vibration(vibration) & TD_phi_Ne_m(phi_Ne_m)
        & TD_disturb(disturb) &
        alpha = vibration + phi_Ne_m + disturb)

    PosDisc_ind: AXIOM
        Alw(TD_phi_Ne_m(phi_Ne_m) =>
        TD_phi(phi) & phi_Ne_m = phi / N )

    AngelOpaqueSect: Axiom
        gamma = 2 * Pi / 2 * n

    NumOpaqueSect: AXIOM
        n = 10
END Optic_Disc

Photo_Cell: THEORY
BEGIN

    IMPORTING Optic_Disc

    delta: real
    alpha: VAR real
```

```
    activation: TD_Formula
    k: VAR Pos
    n: VAR posnat

    Iff_activation: AXIOM
        Alw(EX! k: (position(k) <=> activation))

    Poss_lecture: AXIOM
        delta / 2 <= gamma / 3

    Lect1: AXIOM
        Alw(activation & TD_alpha(alpha) &
            (((EXISTS n: 2 * n * gamma <= alpha
                & alpha <= 2 * n * gamma + delta / 2)
            OR (EXISTS n: 2 * n  * gamma + gamma - delta/2 <= alpha
                & alpha <= 2 * n * gamma + gamma + delta/2)
            OR (EXISTS n: 2 * n * gamma + 2 * gamma - delta / 2 <= alpha
                & alpha <= 2 * n * gamma + 2 * gamma))
        => (position(F) OR position(V))))

    Lect2: AXIOM
        Alw(activation & TD_alpha(alpha) &
            ((EXISTS n: 2 * n * gamma + delta / 2 < alpha
                & alpha < 2 * n * gamma + gamma - delta / 2)
        => position(F)))

    Lect3: AXIOM
        Alw(activation & TD_alpha(alpha) &
            ((EXISTS n: 2 * n * gamma + gamma + delta/2 < alpha
                & alpha < 2 * n * gamma + 2 * gamma - delta/2)
        => position(V)))
END Photo_Cell
```

```
Detector: THEORY
BEGIN

    IMPORTING Photo_Cell

    t, t1, t2: Time
    quantum, insert: TD_Formula
    content_1, content_0, precState_1, precState_0: TD_Formula

    Const_0: AXIOM
        t = 20

    Const_1: AXIOM
        t1 = 25

    Const_2: AXIOM
        t2 = 310

    Cycle: AXIOM
        Alw(Becomes(activation) =>
        Lasts(activation, t1)
            & Futr(NextTime(activation, t2), t1))

    Insertion: AXIOM
        Alw(Becomes(activation) <=> Futr(insert, t))

    Transition1: AXIOM
        Alw(Becomes(content_1) =>
        Becomes(precState_1) & Until_w(precState_1, content_0))

    Transition2: AXIOM
        Alw(Becomes(content_0) =>
        Becomes(precState_0) & Until_w(precState_0, content_1))

    SendQuantum: AXIOM
        Alw(quantum <=>
        Becomes(precState_1) OR Becomes(precState_0))
END Detector
```

```
Shift_register: THEORY
BEGIN

    IMPORTING Detector

    i, j: VAR nat
    eight: nat


    Initial_value: AXIOM
        Alw(AlwP(NOT insert) => FA! i: NOT slot(i))

    Eight: AXIOM
        Alw(eight = 8)

    Contraint_i: AXIOM
        Alw(i <= 8)

    Contraint_j: AXIOM
        Alw(j <= 7)

    Postponement: AXIOM
        Alw(insert =>
        (position(F) => Until_ie(slot(eight), insert)) &
        (position(V) => Until_ie(NOT slot(eight), insert)) &
        (FA! j: ((slot(j+1) => Until_ie(slot(j), insert)) &
            (NOT slot(j+1) => Until_ie(NOT slot(j),insert)))))

    Consistence: AXIOM
        Alw(FA! i: ((NOT insert & slot(i)) => slot(i)))

    Content_0: AXIOM
        Alw(FA! i: (NOT slot(i)) <=> content_0)

    Content_1: AXIOM
        Alw(FA! i: (slot(i)) <=> content_1)
END Shift_register
```

```
Tariff_program: THEORY
BEGIN

    IMPORTING Shift_register

    m, m1: VAR Month
    d, g1: VAR Day
    h, d1, h1: VAR Hour
    f: VAR Hol1
    t1, t2: VAR Tarif
    fluent_tariff: Tarif
    f0, f1, f2, f3, f4, f5: VAR Hour
    F0, F1, F2, F3: VAR Hour


    tariff: FUNCTION
        [[Day, Month, Hour] -> Tarif]

    d_pub_hol: FUNCTION
        [Hol1 -> Hour]

    t_pub_hol: FUNCTION
        [Hol1 -> Tarif]

    d_hol: FUNCTION
        [Hol2 -> Hour]

    t_hol: FUNCTION
        [Hol2 -> Tarif]

    Mobil_pt: AXIOM
        Alw(fluent_tariff = T4 <=> tariff_m(d, m, h))

    Fix: AXIOM
        Alw(NOT tariff_m(d, m, h) =>
        fluent_tariff = tariff(d, m, h))
```

```
Public_holiday: AXIOM
    Alw(public_holiday(d, m) =>
    d_pub_hol(1) + d_pub_hol(2) + d_pub_hol(3) = 48)

Working: AXIOM
    Alw(NOT public_holiday(d, m) =>
    d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4) + d_hol(5) = 48)

Ext_hol_0: AXIOM
    f0 = 0

Ext_hol_1: AXIOM
    f1 = d_hol(1)

Ext_hol_2: AXIOM
    f2 = d_hol(1) + d_hol(2)

Ext_hol_3: AXIOM
    f3 = d_hol(1) + d_hol(2) + d_hol(3)

Ext_hol_4: AXIOM
    f4 = d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4)

Ext_hol_5: AXIOM
    f5 = d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4) + d_hol(5)

Ext_pub_hol_0: AXIOM
    F0 = 0

Ext_pub_hol_1: AXIOM
    F1 = d_pub_hol(1)

Ext_pub_hol_2: AXIOM
    F2 = d_pub_hol(1) + d_pub_hol(2)

Ext_pub_hol_3: AXIOM
    F3 = d_pub_hol(1) + d_pub_hol(2) + d_pub_hol(3)
```

```
    Tariffs: AXIOM
        Alw(tariff(d, m, h) = t1 <=>
        (public_holiday(d, m) &
            (((F0 < h & h < F1) & (t_pub_hol(1) = t1)) OR
            ((F1 < h & h < F2) & (t_pub_hol(2) = t1)) OR
            ((F2 < h & h < F3) & (t_pub_hol(3) = t1)))
        & NOT public_holiday(d, m) &
            (((f0 < h & h < f1) & (t_hol(1) = t1)) OR
            ((f1 < h & h < f2) & (t_hol(2) = t1)) OR
            ((f2 < h & h < f3) & (t_hol(3) = t1)) OR
            ((f3 < h & h < f4) & (t_hol(4) = t1)) OR
            ((f4 < h & h < f5) & (t_hol(5) = t1)))))

    Postponement: AXIOM
        Alw(post_prog =>
        ((FORALL f: FORALL d1: FORALL t2:
        Alw((d_pub_hol_post(f) = d1 =>
        d_pub_hol(f) = d1 & Until(d_pub_hol(f) = d1, post_prog))

        & (t_pub_hol_post(f) = t2 =>
        t_pub_hol(f) = t2 & Until(t_pub_hol(f) = t2, post_prog))

        & (d_hol_post(f) = d1 =>
        d_hol(f) = d1 & Until(d_hol(f) = d1, post_prog))

        & (t_hol_post(f) = t2 =>
        t_hol(f) = t2 & Until(t_hol(f) = t2, post_prog))))

        & (FORALL g1: FORALL m1: FORALL h1:
        Alw((tariff_m_post(g1, m1, h1) =>
        tariff_m(g1, m1, h1) & Until(tariff_m(g1, m1, h1), post_prog))

        & NOT (tariff_m_post(g1, m1, h1) =>
        NOT tariff_m(g1, m1, h1)
        & Until(NOT tariff_m(g1, m1, h1), post_prog))))))
END Tariff_program
```

```
Distributor: THEORY
BEGIN

    IMPORTING Tariff_program

    nq: VAR nat
    n, med_power, inst_power: VAR real
    TD_k: TD_Var[Time, real].TD_var
    k: real
    Delta, t: VAR Time
    t1, t2: VAR Tarif
    ckPot: TD_Formula

    Valid_quantum: AXIOM
        Alw(quantum => energy_weight(fluent_tariff))

    Non_valid_quantum: AXIOM
        NOT EXISTS t1: Alw(NOT quantum => energy_weight(t1))

    Unicity: AXIOM
        EXISTS t1, t2: Alw((energy_weight(t1) & energy_weight(t2)
        => t1 = t2))

    Clock: AXIOM
        Som(ckPot & Lasts(NOT ckPot, Delta)) &
        Alw(ckPot <=> Futr(ckPot, Delta))

    Initial_quantum: AXIOM
        Alw(ckPot & quantum => nq = 1)

    Initial_non_quantum: AXIOM
        Alw(ckPot & NOT quantum => nq = 0)

    Incrementation: AXIOM
        Alw(quantum & EX! t: (LastTime(quantum OR ckPot, t)
        & Past(nq = n, t))
        => nq = n + 1)
```

```
    Med_power: AXIOM
        Alw(ckPot & EX! t: (LastTime(quantum OR ckPot, t)
        & Past(nq = n, t))
        => TD_med_power(med_power) & med_power = n / t
        & Until(med_power = n / t, ckPot))

    Instantiation_power: AXIOM
        Alw(quantum & EX! t: (LastTime(quantum, t)
        => TD_inst_power(inst_power) & TD_k(k)
        & inst_power = k / t))
END Distributor

Totalizer: THEORY
BEGIN

    IMPORTING Distributor

    tot, n, total: VAR nat
    t, t1, t2: VAR Tarif

% Behavior of Counter
    e: VAR [Tarif -> TD_Formula]
    Add, Totn, Totn1: TD_Formula
    TotN: AXIOM
        Alw(Totn => tot = n)
    TotN1: AXIOM
        Alw(Totn1 => tot = n+1)
    AddCounter(e) : TD_Formula =
        UpToNow(Totn) & Add => Until_w(Totn1, Add)
    Counter?(e): bool = Alw(AddCounter(e))
        partial_tot: VAR(Counter?)
% End of Counter

    Totaln, Totaln1: TD_Formula

    TotalN: AXIOM
        Alw(Totaln => TD_total(total) & total = n)
```

```
    TotalN1: AXIOM
        Alw(Totaln1 => TD_total(total) & total = n+1)

    Tot: AXIOM
        EXISTS t1, t2: Alw(UpToNow(Totaln) & energy_weight(t1) =>
        Until_ie(Totaln1, energy_weight(t2)))

    Partial: AXIOM
        Alw(energy_weight(t) <=> partial_tot(t))
END Totalizer
```

C.1.2 The falsification in PVS

```
Falsification: THEORY
BEGIN

    IMPORTING Shift_register

    alpha: VAR real
    ZoneI: TD_Formula
    n: VAR posnat

    A1: AXIOM
        Alw(ZoneI & TD_alpha(alpha)
        <=>((EXISTS n: 2 * n * gamma <= alpha
            & alpha <= 2 * n * gamma + delta / 2)
        OR (EXISTS n: 2 * n * gamma + gamma - delta/2 <= alpha
            & alpha <= 2 * n * gamma + gamma + delta/2)
        OR (EXISTS n: 2 * n * gamma + 2 * gamma - delta / 2 <= alpha
            & alpha <= 2 * n * gamma + 2 * gamma)))

    L1: LEMMA
        Alw((ZoneI & TD_alpha(alpha) & activation)
        => position(V) OR position(F))

    L2: LEMMA
        Alw(((position(V) & insert) OR (position(F) & insert))
```

```
        => content_0 OR content_1)

    L3: LEMMA
        Alw((UpToNow(NOT content_0) & content_0)
            OR (UpToNow(NOT content_1) & content_1)
        => quantum)

    Falsif: THEOREM
        Alw((ZoneI & TD_alpha(alpha) & activation
        => (content_0 OR content_1))
        & ((UpToNow(NOT content_0) & content_0) => quantum)
        & ((UpToNow(NOT content_1) & content_1) => quantum))
END Falsification
```

C.2 The Energy_Meter with two photocells

In this section, we first represent all the PVS files related to the translation of the TRIO+ specification of the Energy_Meter with two photocells into TRIO/PVS. Further, we show the PVS files of the verification in section C.2.2. In the last section, the PVS files of the different properties are represented. As in the case of the Energy_Meter with one photocell, all the different lemmas and theorems are proved, but the different proof-steps are not represented in this document.

C.2.1 The specification in PVS

```
Energy_Meter_2: THEORY
BEGIN

    IMPORTING trio[int]

% Types
    Time: TYPE = int
    Pos: TYPE = {F, V}
    Month: TYPE = {n: posnat | 1 <= n & n <= 31} CONTAINING 1
    Day: TYPE = {n: posnat | 1 <= n & n <= 12} CONTAINING 1
    Hour: TYPE = {n: posnat | 1 <= n & n <= 48} CONTAINING 1
```

```
    Tarif: TYPE = {T1, T2, T3, T4}
    Hol1: TYPE = {n: posnat | 1 <= n & n <= 3} CONTAINING 1
    Hol2: TYPE = {n: posnat | 1 <= n & n <= 5} CONTAINING 1
% End Types

    IMPORTING TD_Var[Time, Pos]
    IMPORTING TD_Var[Time, [Day, Month, Hour]]
    IMPORTING TD_Var[Time, [Day, Month]]
    IMPORTING TD_Var[Time, Tarif]
    IMPORTING TD_Var[Time, int]

% TD Vars
    position, position1, position2: TD_Var[Time, Pos].TD_var
    slot: TD_Var[Time,nat].TD_var
    tariff_m_post, tariff_m: TD_Var[Time, [Day, Month, Hour]].TD_var
    public_holiday: TD_Var[Time, [Day, Month]].TD_var
    energy_weight: TD_Var[Time, Tarif].TD_var

    TD_med_power: TD_Var[Time, real].TD_var
    TD_inst_power: TD_Var[Time, real].TD_var
    TD_disturb: TD_Var[Time, real].TD_var
    TD_phi: TD_Var[Time, real].TD_var
    TD_vibration: TD_Var[Time, real].TD_var
    TD_phi_Ne_m: TD_Var[Time, real].TD_var
    TD_alpha: TD_Var[Time, real].TD_var
    TD_shift: TD_Var[Time, real].TD_var

    TD_total: TD_Var[Time, int].TD_var
% End TD Vars

    m_power: real
    post_prog: TD_Formula

    d_pub_hol_post: FUNCTION
        [Hol1 -> Hour]

    t_pub_hol_post: FUNCTION
        [Hol1 -> Tarif]
```

```
    d_hol_post: FUNCTION
        [Hol2 -> Hour]

    t_hol_post: FUNCTION
        [Hol2 -> Tarif]
END Energy_Meter_2

G_Ferraris_2: THEORY
BEGIN

    IMPORTING Energy_Meter_2

    phi_epsilon: real
    med_power, vibration: VAR real
    x, V, A, c: VAR real
    start: TD_Formula
    t: VAR Time

    Sinus: FUNCTION
        [real -> real]

    Sin1: AXIOM
        Alw((Sinus(x) >= -1) &
        (Sinus(x) <= 1) & (Sinus(0) =0))

    Vibr: AXIOM
        Alw(TD_med_power(med_power)
        & V = A * Sinus(c * med_power * t))

    Initial: AXIOM
        Alw(Past(start, t) => TD_vibration(vibration) & vibration = V)

    Initial_vibration: AXIOM
        Alw(TD_vibration(vibration) & (vibration) <= phi_epsilon)
END G_Ferraris_2
```

```
Optic_Disc_2: THEORY
BEGIN

    IMPORTING G_Ferraris_2

    Pi: real
    n, N, gamma: posnat
    alpha1, alpha2, vibration, phi, phi_Ne_m, disturb, shift: VAR real

    Plus3: AXIOM
        Alw(TD_alpha(alpha1) =>
        TD_vibration(vibration) & TD_phi_Ne_m(phi_Ne_m)
        & TD_disturb(disturb) &
        alpha1 = vibration + phi_Ne_m + disturb)

    Plus2: AXIOM
        Alw(TD_alpha(alpha2) =>
        TD_alpha(alpha1) & TD_shift(shift)
        & alpha2 = alpha1 + shift)

    PosDisc_ind: AXIOM
        Alw(TD_phi_Ne_m(phi_Ne_m) =>
        TD_phi(phi) & phi_Ne_m = phi / N)

    AngelOpaqueSect: Axiom
        gamma = 2 * Pi / 2 * n

    NumOpaqueSect: AXIOM
        n = 10
END Optic_Disc_2

Photo_Cell_2: THEORY
BEGIN

    IMPORTING Optic_Disc_2

    delta: real
    activation: TD_Formula
```

```
alpha1, alpha2: VAR real
k: VAR Pos
n: VAR posnat

Iff_activation: AXIOM
    Alw(EX! k: (position(k) <=> activation))

Poss_lecture: AXIOM
    delta / 2 <= gamma / 3

Lect11: AXIOM
    Alw(activation & TD_alpha(alpha1) &
    (((EXISTS n: 2 * n * gamma <= alpha1
        & alpha1 <= 2 * n * gamma + delta / 2)
      OR (EXISTS n: 2 * n  * gamma + gamma - delta/2 <= alpha1
        & alpha1 <= 2 * n * gamma + gamma + delta/2)
      OR (EXISTS n: 2 * n * gamma + 2 * gamma - delta / 2 <= alpha1
        & alpha1 <= 2 * n * gamma + 2 * gamma))
    => (position1(F) OR position1(V))))

Lect21: AXIOM
    Alw(activation & TD_alpha(alpha1) &
        ((EXISTS n: 2 * n * gamma + delta / 2 < alpha1
            & alpha1 < 2 * n * gamma + gamma - delta / 2)
    => position1(F)))

Lect31: AXIOM
    Alw(activation & TD_alpha(alpha1) &
        ((EXISTS n: 2 * n * gamma + gamma + delta/2 < alpha1
            & alpha1 < 2 * n * gamma + 2 * gamma - delta/2)
    => position1(V)))

Lect12: AXIOM
    Alw(activation & TD_alpha(alpha2) &
    (((EXISTS n: 2 * n * gamma <= alpha2
        & alpha2 <= 2 * n * gamma + delta / 2)
      OR (EXISTS n: 2 * n  * gamma + gamma - delta/2 <= alpha2
        & alpha2 <= 2 * n * gamma + gamma + delta/2)
```

```
          OR (EXISTS n: 2 * n * gamma + 2 * gamma - delta / 2 <= alpha2
            & alpha2 <= 2 * n * gamma + 2 * gamma))
        => (position2(F) OR position2(V))))

    Lect22: AXIOM
        Alw(activation & TD_alpha(alpha2) &
            ((EXISTS n: 2 * n * gamma + delta / 2 < alpha2
                & alpha2 < 2 * n * gamma + gamma - delta / 2)
        => position2(F)))

    Lect32: AXIOM
        Alw(activation & TD_alpha(alpha2) &
            ((EXISTS n: 2 * n * gamma + gamma + delta/2 < alpha2
                & alpha2 < 2 * n * gamma + 2 * gamma - delta/2)
        => position2(V)))
END Photo_Cell_2

Detector_2: THEORY
BEGIN

    IMPORTING Photo_Cell_2

    t, t1, t2: Time
    quantum, insert: TD_Formula
    content_1, content_0, precState_1, precState_0: TD_Formula
    bup1, bdw1, bup2, bdw2, bupconf1, bdwconf1: TD_Formula

    Const_0: AXIOM
        t = 20

    Const_1: AXIOM
        t1 = 25

    Const_2: AXIOM
        t2 = 310
```

```
    Cycle: AXIOM
        Alw(Becomes(activation) =>
        Lasts(activation, t1) & Futr(NextTime(activation, t2), t1))

    BeginUp1: AXIOM
        Alw(bup1 <=> position1(F)
            & Since(NOT position1(F), position1(V)))

    BeginDw1: AXIOM
        Alw(bdw1 <=> position1(V)
            & Since(NOT position1(V), position1(F)))

    BeginUp2: AXIOM
        Alw(bup2 <=> position2(F)
            & Since(NOT position2(F), position2(V)))

    BeginDw2: AXIOM
        Alw(bdw2 <=> position2(V)
            & Since(NOT position2(V), position2(F)))

    BeginUpCf1: AXIOM
        Alw(bupconf1 <=> bup2 & Since(NOT bup2, bup1))

    BeginDwCf1: AXIOM
        Alw(bdwconf1 <=> bdw2 & Since(NOT bdw2, bdw1))

    SendQuantum: AXIOM
        Alw(quantum <=> (bupconf1 OR bdwconf1))
END Detector_2

Tariff_program_2: THEORY
BEGIN

    IMPORTING Detector_2

    m, m1: VAR Month
    d, g1: VAR Day
    h, d1, h1: VAR Hour
```

```
f: VAR Hol1
f_tariff, t, t1: VAR Tarif
fluent_tariff: Tarif
f0, f1, f2, f3, f4, f5: VAR Hour
F0, F1, F2, F3: VAR Hour

tariff: FUNCTION
    [Day, Month, Hour -> Tarif]

d_pub_hol: FUNCTION
    [Hol1 -> Hour]

t_pub_hol: FUNCTION
    [Hol1 -> Tarif]

d_hol: FUNCTION
    [Hol2 -> Hour]

t_hol: FUNCTION
    [Hol2 -> Tarif]

Mobil_pt: AXIOM
    Alw(fluent_tariff = T4 <=> tariff_m(d, m, h))

Fix: AXIOM
    Alw(NOT tariff_m(d, m, h) =>
    fluent_tariff = tariff(d, m, h))

Public_holiday: AXIOM
    Alw(public_holiday(d, m) =>
    d_pub_hol(1) + d_pub_hol(2) + d_pub_hol(3) = 48)

Working: AXIOM
    Alw(NOT public_holiday(d, m) =>
    d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4) + d_hol(5) = 48)

Ext_hol_0: AXIOM
    f0 = 0
```

```
Ext_hol_1: AXIOM
    f1 = d_hol(1)

Ext_hol_2: AXIOM
    f2 = d_hol(1) + d_hol(2)

Ext_hol_3: AXIOM
    f3 = d_hol(1) + d_hol(2) + d_hol(3)

Ext_hol_4: AXIOM
    f4 = d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4)

Ext_hol_5: AXIOM
    f5 = d_hol(1) + d_hol(2) + d_hol(3) + d_hol(4) + d_hol(5)

Ext_pub_hol_0: AXIOM
    F0 = 0

Ext_pub_hol_1: AXIOM
    F1 = d_pub_hol(1)

Ext_pub_hol_2: AXIOM
    F2 = d_pub_hol(1) + d_pub_hol(2)

Ext_pub_hol_3: AXIOM
    F3 = d_pub_hol(1) + d_pub_hol(2) + d_pub_hol(3)

Tariffs: AXIOM
    Alw(tariff(d, m, h) = t <=>
    (public_holiday(d, m) &
        (((F0 < h & h < F1) & (t_pub_hol(1) = t)) OR
            ((F1 < h & h < F2) & (t_pub_hol(2) = t)) OR
            ((F2 < h & h < F3) & (t_pub_hol(3) = t)))
    & NOT public_holiday(d, m) &
        (((f0 < h & h < f1) & (t_hol(1) = t)) OR
        ((f1 < h & h < f2) & (t_hol(2) = t)) OR
        ((f2 < h & h < f3) & (t_hol(3) = t)) OR
```

```
            ((f3 < h & h < f4) & (t_hol(4) = t)) OR
            ((f4 < h & h < f5) & (t_hol(5) = t)))))

    postponement: AXIOM
        Alw(post_prog =>
        (FORALL f: FORALL d1: FORALL t1:

        Alw((d_pub_hol_post(f) = d1 =>
        d_pub_hol(f) = d1 & Until(d_pub_hol(f) = d1, post_prog))

        & (t_pub_hol_post(f) = t1 =>
        t_pub_hol(f) = t1 & Until(t_pub_hol(f) = t1, post_prog))

        & (d_hol_post(f) = d1 =>
        d_hol(f) = d1 & Until(d_hol(f) = d1, post_prog))

        & (t_hol_post(f) = t1 =>
        t_hol(f) = t1 & Until(t_hol(f) = t1, post_prog))))

        & FORALL g1: FORALL m1: FORALL h1:

        Alw((tariff_m_post(g1, m1, h1) =>
        tariff_m(g1, m1, h1) & Until(tariff_m(g1, m1, h1), post_prog))

        & NOT (tariff_m_post(g1, m1, h1) =>
        NOT tariff_m(g1, m1, h1) &
        Until(NOT tariff_m(g1, m1, h1), post_prog))))
END Tariff_program_2

Distributor_2: THEORY
BEGIN

    IMPORTING Tariff_program_2

    nq: VAR nat
    n, med_power, inst_power: VAR real
    TD_k: TD_Var[Time, real].TD_var
    k: real
```

```
Delta, t: VAR Time
t1, t2: VAR Tarif
ckPot: TD_Formula

Valid_quantum: AXIOM
    Alw(quantum => energy_weight(fluent_tariff))

Non_valid_quantum: AXIOM
    NOT EXISTS t1: Alw(NOT quantum => energy_weight(t1))

Unicity: AXIOM
    EXISTS t1, t2: Alw(energy_weight(t1) & energy_weight(t2)
    => t1 = t2)

Clock: AXIOM
    Som(ckPot & Lasts(NOT ckPot, Delta)) &
    Alw(ckPot <=> Futr(ckPot, Delta))

Initial_quantum: AXIOM
    Alw(ckPot & quantum => nq = 1)

Initial_non_quantum: AXIOM
    Alw(ckPot & NOT quantum => nq = 0)

Incrementation: AXIOM
    Alw(quantum & EX! t: (LastTime(quantum OR ckPot, t)
    & Past(nq = n, t))
    => nq = n + 1)

Med_power: AXIOM
    Alw(ckPot & EX! t: (LastTime(quantum OR ckPot, t)
    & Past(nq = n, t))
    => TD_med_power(med_power) & med_power = n / t
    & Until(med_power = n / t, ckPot))
```

```
    Instantiation_power: AXIOM
        Alw(quantum & EX! t: (LastTime(quantum, t)
        => TD_inst_power(inst_power) & TD_k(k)
        & inst_power = k / t))
END Distributor_2

Totalizer_2: THEORY
BEGIN

    IMPORTING Distributor_2

    total, tot, n: VAR nat
    t, t1, t2: VAR Tarif

% Behaviour of Counter
    e: VAR [Tarif -> TD_Formula]
    Add, Totn, Totn1: TD_Formula
    TotN: AXIOM
        Alw(Totn => tot = n)
    TotN1: AXIOM
        Alw(Totn1 => tot = n+1)
    AddCounter(e) : TD_Formula =
        UpToNow(Totn) & Add => Until_w(Totn1, Add)
    Counter?(e): bool = Alw(AddCounter(e))
    partial_tot: VAR(Counter?)
% End of Counter

    Totaln, Totaln1: TD_Formula

    TotalN: AXIOM
        Alw(Totaln => TD_total(total) & total = n)

    TotalN1: AXIOM
        Alw(Totaln1 => TD_total(total) & total = n+1)

    Tot: AXIOM
        EXISTS t1, t2: Alw(UpToNow(Totaln) & energy_weight(t1) =>
        Until_ie(Totaln1, energy_weight(t2)))
```

```
    Partial: AXIOM
        Alw(energy_weight(t) <=> partial_tot(t))
END Totalizer_2
```

C.2.2 The verification in PVS

```
Verification: THEORY
BEGIN
    IMPORTING Detector_2

    alpha, alpha1, alpha2: VAR real
    ZoneI1, ZoneI2, ZoneF1, ZoneF2, ZoneV1, ZoneV2: TD_Formula

    A11: LEMMA
        Alw((ZoneI1 & TD_alpha(alpha1) & activation)
        => position1(F) OR position1(V))

    A12: LEMMA
        Alw((ZoneI2 & TD_alpha(alpha2) & activation)
        => position2(F) OR position2(V))

    A21: LEMMA
        Alw((ZoneF1 & TD_alpha(alpha1) & activation)
        => position1(F))

    A22: LEMMA
        Alw((ZoneF2 & TD_alpha(alpha2) & activation)
        => position2(F))

    A31: LEMMA
        Alw((ZoneV1 & TD_alpha(alpha1) & activation)
        => position1(V))

    A32: LEMMA
        Alw((ZoneV2 & TD_alpha(alpha2) & activation)
        => position2(V))
```

```
A41: AXIOM
    Alw(Since(NOT bup2, bup1) <=>
    Since(NOT(position2(F)
        & Since(NOT position2(F), position2(V))),
        (position1(F)
        & Since(NOT position1(F), position1(V)))))

A42: AXIOM
    Alw(Since(NOT bdw2, bdw1) <=>
    Since(NOT(position2(V)
        & Since(NOT position2(V), position2(F))),
        (position1(V)
        & Since(NOT position1(V), position1(F)))))


Quantum: THEOREM
    Alw(
    (((position2(F) & Since(NOT position2(F), position2(V)))
    & Since (NOT
      (position2(F) & Since(NOT position2(F), position2(V))),
      (position1(F) & Since(NOT position1(F), position1(V)))))
                          OR
    ((position2(V) & Since(NOT position2(V), position2(F)))
    & Since (NOT
      (position2(V) & Since(NOT position2(V), position2(F))),
      (position1(V) & Since(NOT position1(V), position1(F))))))
                          <=>
                       quantum)

T2: THEOREM
    Alw((((position2(F) & Since(NOT position2(F), position2(V))
        & Since(NOT(position2(F)
        & Since(NOT position2(F), position2(V))),
        (position1(F)
        & Since(NOT position1(F), position1(V)))))
    & ((position1(F) & Since(NOT position1(F), position1(V)))
      OR ((position1(V) & Since(NOT position1(V), position1(F)))
       & (Since(NOT position1(F), position1(V))))))
```

```
        OR
          ((position2(V) & Since(NOT position2(V), position2(F))
            & Since(NOT(position2(V)
            & Since(NOT position2(V), position2(F))),
            (position1(V)
            & Since(NOT position1(V), position1(F)))))
         & ((position1(V) & Since(NOT position1(V), position1(F)))
           OR ((position1(F) & Since(NOT position1(F), position1(V)))
            & (Since(NOT position1(V), position1(F)))))))
         => quantum)
END Verification

Verif1: THEORY
BEGIN
    IMPORTING Detector_2

    alpha, alpha1, alpha2: VAR real
    ZoneI1, ZoneI2, ZoneF1, ZoneF2, ZoneV1, ZoneV2: TD_Formula
    n: VAR posnat

    A11: AXIOM
        Alw(ZoneI1 & TD_alpha(alpha1)
        <=>((EXISTS n: 2 * n * gamma <= alpha1
            & alpha1 <= 2 * n * gamma + delta / 2)
          OR (EXISTS n: 2 * n * gamma + gamma - delta/2 <= alpha1
            & alpha1 <= 2 * n * gamma + gamma + delta/2)
          OR (EXISTS n: 2 * n * gamma + 2 * gamma - delta / 2 <= alpha1
            & alpha1 <= 2 * n * gamma + 2 * gamma))
        & NOT(ZoneV1 OR ZoneF1))

    A21: AXIOM
        Alw(ZoneF1 & TD_alpha(alpha1)
        <=> (EXISTS n: 2 * n * gamma + delta / 2 < alpha1
            & alpha1 < 2 * n * gamma + gamma - delta / 2)
        & NOT (ZoneI1 OR ZoneV1))
```

```
A31: AXIOM
    Alw(ZoneV1 & TD_alpha(alpha1)
    <=> (EXISTS n: 2 * n * gamma + gamma + delta/2 < alpha1
        & alpha1 < 2 * n * gamma + 2 * gamma - delta/2)
    & NOT (ZoneI1 OR ZoneF1))

A12: AXIOM
    Alw(ZoneI2 & TD_alpha(alpha2)
    <=>((EXISTS n: 2 * n * gamma <= alpha2
        & alpha2 <= 2 * n * gamma + delta / 2)
      OR (EXISTS n: 2 * n * gamma + gamma - delta/2 <= alpha2
        & alpha2 <= 2 * n * gamma + gamma + delta/2)
      OR (EXISTS n: 2 * n * gamma + 2 * gamma - delta / 2 <= alpha2
        & alpha2 <= 2 * n * gamma + 2 * gamma))
    & NOT(ZoneV2 OR ZoneF2))

A22: AXIOM
    Alw(ZoneF2 & TD_alpha(alpha2)
    <=> (EXISTS n: 2 * n * gamma + delta / 2 < alpha2
        & alpha2 < 2 * n * gamma + gamma - delta / 2)
    & NOT (ZoneI2 OR ZoneV2))

A32: AXIOM
    Alw(ZoneV2 & TD_alpha(alpha2)
    <=> (EXISTS n: 2 * n * gamma + gamma + delta/2 < alpha2
        & alpha2 < 2 * n * gamma + 2 * gamma - delta/2)
    & NOT (ZoneI2 OR ZoneF2))

L11: LEMMA
    Alw(ZoneI1 & TD_alpha(alpha1) & activation
    => position1(F) OR position1(V))

L21: LEMMA
    Alw(ZoneF1 & TD_alpha(alpha1) & activation
    => position1(F))
```

```
    L31: LEMMA
        Alw(ZoneV1 & TD_alpha(alpha1) & activation
        => position1(V))

    L12: LEMMA
        Alw(ZoneI2 & TD_alpha(alpha2) & activation
        => position2(F) OR position2(V))

    L22: LEMMA
        Alw(ZoneF2 & TD_alpha(alpha2) & activation
        => position2(F))

    L32: LEMMA
        Alw(ZoneV2 & TD_alpha(alpha2) & activation
        => position2(V))
END Verif1

Verif2: THEORY
BEGIN

    IMPORTING Verif1

    alpha1: VAR real
    old_pos, new_pos: VAR real
    ti: VAR Time
    n: VAR posnat

    A1: AXIOM
        Alw(quantum & TD_alpha(alpha1)
        => (old_pos = alpha1))

    A2: AXIOM
        Alw(quantum & TD_alpha(alpha1)
        <=> (NextTime(quantum, ti) & Lasts(NOT quantum, ti))
            & ti > 0 & Futr(TD_alpha(alpha1), ti))
```

```
    L1: LEMMA
        Alw(NextTime(quantum, ti) & Futr(TD_alpha(alpha1), ti)
        => (new_pos = alpha1))

    L2: LEMMA
        Alw((ZoneF1 & TD_alpha(alpha1) & quantum)
        => (EXISTS n: (2 * n * gamma + delta/2 < old_pos)
            & (old_pos < 2 * n * gamma + gamma - delta/2))
         & NOT(ZoneI1 OR ZoneV1))

    L3: LEMMA
        Alw((Futr(ZoneV1, ti) & Futr(TD_alpha(alpha1), ti)
            & NextTime(quantum, ti))
        => (EXISTS n: (2 * n * gamma + gamma + delta/2 < new_pos)
            & (new_pos < 2 * n * gamma + 2 * gamma - delta/2))
         & Futr(NOT(ZoneI1 OR ZoneF1), ti))

    T1: THEOREM
        Alw((Futr((ZoneF1 & TD_alpha(alpha1) & quantum), ti)
            & (Futr(ZoneV1, ti) & Futr(TD_alpha(alpha1), ti)
            & NextTime(quantum, ti)))
        => (old_pos + (gamma - delta) <= new_pos)
         & (old_pos + (gamma + delta) >= new_pos))

END Verif2
```

C.2.3 The different properties in PVS

```
Properties: THEORY
BEGIN

    IMPORTING Verification

    alpha1, alpha2: VAR real

    A1F: AXIOM
        Alw(NOT position1(F) <=> position1(V))
```

```
A1V: AXIOM
    Alw(NOT position1(V) <=> position1(F))

A2F: AXIOM
    Alw(NOT position2(F) <=> position2(V))

A2V: AXIOM
    Alw(NOT position2(V) <=> position2(F))

A2Dw: AXIOM
    Alw(position2(V) & Past(position2(V), t1)
    => NOT Since(NOT position2(V), position2(F)))

A2Up: AXIOM
    Alw(position2(F) & Past(position2(F), t1)
    => NOT Since(NOT position2(F), position2(V)))

L1: LEMMA
    Alw(NOT (bupconf1 OR bdwconf1) => NOT quantum)

L2: LEMMA
    Alw(NOT bup2 OR NOT Since(NOT bup2, bup1) => NOT bupconf1)

L3: LEMMA
    Alw(NOT bdw2 OR NOT Since(NOT bdw2, bdw1) => NOT bdwconf1)

L4: LEMMA
    Alw(NOT position1(F)
        OR NOT Since(NOT position1(F), position1(V))
    => NOT bup1)

L5: LEMMA
    Alw(NOT position2(F)
        OR NOT Since(NOT position2(F), position2(V))
    => NOT bup2)
```

```
L6: LEMMA
    Alw(NOT position1(V)
        OR NOT Since(NOT position1(V), position1(F))
    => NOT bdw1)

L7: LEMMA
    Alw(NOT position2(V)
        OR NOT Since(NOT position2(V), position2(F))
    => NOT bdw2)

LP111: LEMMA
    Alw(ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
    => position1(V) & position2(V)
       & Past(position1(V) & position2(V), t1))

LP112: LEMMA
    Alw(position1(V) & position2(V)
    & Past(position1(V) & position2(V), t1)
    => NOT(bup2 OR bdw2))

LP113: LEMMA
    Alw(NOT(bup2 OR bdw2) => NOT quantum)

P11: THEOREM
    Alw(ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
    => NOT quantum)

LP121: LEMMA
    Alw(ZoneF1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneF1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
```

```
        => position1(F) & position2(F)
           & Past(position1(F) & position2(F), t1))

    LP122: LEMMA
        Alw(position1(F) & position2(F)
        & Past(position1(F) & position2(F), t1)
        => NOT(bdw2 OR bup2))

    LP123: LEMMA
        Alw(NOT(bdw2 OR bup2) => NOT quantum)

    P12: THEOREM
        Alw(ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)
        & Past((ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
        => NOT quantum)
END Properties

Prop2: THEORY
BEGIN

    IMPORTING Properties

    alpha1, alpha2: VAR real

    A3: AXIOM
        Alw(position2(V) & Past(position2(F), t1)
        & Past(position2(F), t1 + t1)
        => Since(NOT position2(V), position2(F)))

    A4: AXIOM
        Alw(Past(position1(V), t1)
        & Past(position1(F), t1 + t1)
        => Past(Since(NOT position1(V), position1(F)), t1))

    A5: AXIOM
        Alw(position2(V) & Since(NOT position2(V), position2(F))
```

```
    & Past(position1(V)
        & Since(NOT position1(V), position1(F)), t1)
    => Since(NOT
      (position2(V) & Since(NOT position2(V), position2(F))),
      (position1(V) & Since(NOT position1(V), position1(F))))

LP211: LEMMA
    Alw(ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneV1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
    & Past((ZoneF1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
    => position1(V) & position2(V)
       & Past((position1(V) & position2(F)), t1)
       & Past((position1(F) & position2(F)), t1 + t1))

LP212: LEMMA
    Alw(position1(V) & position2(V)
    & Past((position1(V) & position2(F)), t1)
    & Past((position1(F) & position2(F)), t1 + t1)
    => ((position2(V) & Since(NOT position2(V), position2(F)))
        & Since(NOT
          (position2(V) & Since(NOT position2(V), position2(F))),
          (position1(V) & Since(NOT position1(V), position1(F))))))

P21: THEOREM
    Alw(ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneV1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
    & Past((ZoneF1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
    => quantum)
```

```
A6: AXIOM
    Alw(position2(F) & Past(position2(V), t1)
    & Past(position2(V), t1 + t1)
    => Since(NOT position2(F), position2(V)))

A7: AXIOM
    Alw(Past(position1(F), t1)
    & Past(position1(V), t1 + t1)
    => Past(Since(NOT position1(F), position1(V)), t1))

A8: AXIOM
    Alw(position2(F) & Since(NOT position2(F), position2(V))
    & Past(position1(F)
        & Since(NOT position1(F), position1(V)), t1)
    => Since(NOT
      (position2(F) & Since(NOT position2(F), position2(V))),
      (position1(F) & Since(NOT position1(F), position1(V)))))

LP221: LEMMA
    Alw(ZoneF1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneF1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
    & Past((ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
    => position1(F) & position2(F)
       & Past((position1(F) & position2(V)), t1)
       & Past((position1(V) & position2(V)), t1 + t1))

LP222: LEMMA
    Alw(position1(F) & position2(F)
    & Past((position1(F) & position2(V)), t1)
    & Past((position1(V) & position2(V)), t1 + t1)
    => ((position2(F) & Since(NOT position2(F), position2(V)))
        & Since(NOT
          (position2(F) & Since(NOT position2(F), position2(V))),
          (position1(F) & Since(NOT position1(F), position1(V))))))
```

```
    P22: THEOREM
        Alw(ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)
        & Past((ZoneF1 & ZoneV2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
        & Past((ZoneV1 & ZoneV2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
        => quantum)
END Prop2

Prop3: THEORY
BEGIN

    IMPORTING Prop2

    alpha1, alpha2: VAR real

    LP31: LEMMA
        Alw(ZoneV1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)
        & Past((ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
        => position1(V) & position2(F)
           & Past((position1(F) & position2(F)), t1))

    LP32: LEMMA
        Alw(position1(V) & position2(F)
        & Past((position1(F) & position2(F)), t1)
        => NOT(bdw2 OR bup2))

    P3: THEOREM
        Alw(ZoneV1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)
        & Past((ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
        => NOT quantum)
END Prop3
```

```
Prop4: THEORY
BEGIN

    IMPORTING Prop3

    alpha1, alpha2: VAR real

    A91: AXIOM
        Alw(position2(V) & Past(position2(V), t1)
        & Past(position2(F), t1 + t1)
        => Past(Since(NOT position2(V), position2(F)), t1))

    A92: AXIOM
        Alw(Past(position1(V), t1) & Past(position1(F), t1 + t1)
        => Past(Since(NOT position1(V), position1(F)), t1))

    A10: AXIOM
        Alw(position2(V)
        & Past(Since(NOT position2(V), position2(F)), t1)
        & Past(position1(V)
            & Since(NOT position1(V), position1(F)), t1)
        => Past(Since(NOT
          (position2(V) & Since(NOT position2(V), position2(F))),
          (position1(V) & Since(NOT position1(V), position1(F)))), t1))

    LP411: LEMMA
        Alw(ZoneV1 & ZoneV2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)
        & Past((ZoneV1 & ZoneI2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
        & Past((ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
        => position1(V) & position2(V)
           & Past((position1(V) & (position2(V) OR position2(F))), t1)
           & Past((position1(F) & position2(F)), t1 + t1))
```

```
LP412: LEMMA
    Alw(position1(V) & position2(V)
    & Past((position1(V) & position2(F)), t1)
    & Past((position1(F) & position2(F)), t1 + t1)
    => quantum)

LP413: LEMMA
    Alw(position1(V) & position2(V)
    & Past((position1(V) & position2(V)), t1)
    & Past((position1(F) & position2(F)), t1 + t1)
    => Past(quantum, t1))

P41: THEOREM
    Alw(ZoneV1 & ZoneV2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)
    & Past((ZoneV1 & ZoneI2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
    & Past((ZoneF1 & ZoneF2 & activation
        & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
    => quantum OR Past(quantum, t1))

LP421: LEMMA
    Alw(position1(V) & position2(V)
    & Past((position1(V) & position2(V)), t1)
    & Past((position1(F) & position2(F)), t1 + t1)
    => NOT quantum)

LP422: LEMMA
    Alw(position1(V) & position2(V)
    & Past((position1(V) & position2(F)), t1)
    & Past((position1(F) & position2(F)), t1 + t1)
    => NOT(Past(quantum, t1)))
```

```
    P42: THEOREM
        Alw(ZoneV1 & ZoneV2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)
        & Past((ZoneV1 & ZoneI2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1)
        & Past((ZoneF1 & ZoneF2 & activation
            & TD_alpha(alpha1) & TD_alpha(alpha2)), t1 + t1)
        => NOT(quantum & Past(quantum, t1)))
END Prop4
```